DESARROLLO DE UN SISTEMA DE INDEXACION Y BUSQUEDA SOBRE LA BASE DE DATOS DE BIOMEDICINA MEDLINE

SISTEMAS INFORMATICOS
FACULTAD DE INFORMATICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Directores: Mónica Chagoyen Quiles Autores: Fabián Fernández García
Moisés Azancot Chocrón

curso 2006-2007

1. INTRODUCCION

1.1 Resumen

Nuestro proyecto consiste en crear dos sistemas de indexación y búsqueda sobre la base de datos de biomedicina MEDLINE, con dos tecnologías distintas de manera que se pueda evaluar cual de ellas es la más adecuada para tratar con MEDLINE.
MEDLINE es la mayor base de datos de referencias bibliograficas en el área biomédica, tiene mas de 15 millones de referencias recopiladas por la Librería Nacional de Medicina Estadounidense (NML) desde el año 1965. MEDLINE se distribuye en formato XML y ocupa alrededor de 55 GB, lo cual hace imprescindible una evaluación del rendimiento a la hora de elegir una tecnología para manejar los datos.
Las tecnologías elegidas son Lucene y PostgreSQL. PostgreSQL es un sistema de bases de datos relacional similar MySQL o Oracle, pero suele ser mas robusta que estas cuando trata con bases de datos muy grandes y Lucene es una Scalable Information Retrieval Library (IR) implementada en java y parte de la familia de proyectos de Apache Jakarta, es decir, Lucene crea bases de datos totalmente textuales y permite realizar búsquedas sobre ellas.
Para insertar los datos en la base de datos es necesario procesar los XML de MEDLINE y extraer la información que nos interesa. Para tal fin hemos utilizado SAX que es una API para analizar datos pero que necesita una implementación de un parser para poder funcionar. Xerces implementa un parser para SAX que funciona bastante bien por lo que hemos elegido este parser para completar SAX, Xerces pertenece al igual que Lucene a la familia de proyectos de Apache Jakarta.
En este documento intentaremos explicar detalladamente el funcionamiento de las tecnologías utilizadas y descartadas, y de los sistemas que hemos implementado, así como una evaluación/comparativa para decidir que sistema es mas adecuado para MEDLINE.
Por ultimo para comunicar nuestra aplicación java de indexación para postgresql y el servidor de postgresql utilizaremos la versión de la librería JDBC implementada especialmente para postgresql.

1.2 Summary

The current project consists of creating two indexing and searching systems over the biomedicine database MEDLINE, with two different technologies in order to evaluate which one of them is more suited to deal with MEDLINE. MEDLINE is the largest database of bibliography references in the biomedical area; it has more than fifteen millions of references collected by the United States National Medicine Library (NML) since year 1965. MEDLINE is distributed in XML format and its size is about fifty five Gigabytes, which makes an evaluation of the performance of each technology definitely essential to make a proper choice to handle the data processing.

The chosen technologies are Lucene and PostgreSQL. PostgreSQL is a system of relational databases similar to MySQL or Oracle, but tends to be stronger when processing massive databases. Lucene is one Scalable Information Retrieval Library (IR) implemented using Java and part of the projects family known as Apache Jakarta, that is to say Lucene create fully textual databases and allows performing searches through them.

To insert data into the database, it is necessary to process the XML files of MEDLINE and extract the information required. To fulfil that purpose we have used SAX, an API used to analyze data but with an implementation of a parser needed in order to work successfully. Xerces implements a parser for SAX which works properly in our case, and so it has been chosen to complete SAX. Xerces belongs, as well as Lucene does, to the projects family known as Apache Jakarta.

In order to communicate our indexing Java application for PostgreSQL and the PostgreSQL's server we will use a JDBC library version specially implemented for PostgreSQL.

Through this document, we will try to explain in great detail the different ways of operation of the chosen and ruled out technologies, the systems which have been implemented, as well as an evaluation/comparison to make a decision about which system is more suitable to MEDLINE.

1.3 Objetivos

Existen muchas implementaciones de buscadores e indexadores, como por ejemplo PubMED, pero la capacidad de búsqueda de éstos es bastante limitada, puesto que solo suelen incluir búsquedas booleanas. Así pues se pretende mejorar en algunos aspectos los sistemas ya existentes:

1) Elegir 2 tecnologías candidatas para manejar los datos de MEDLINE.

2) Indexar los datos interesantes partiendo de los archivos XML que distribuye MEDLINE.

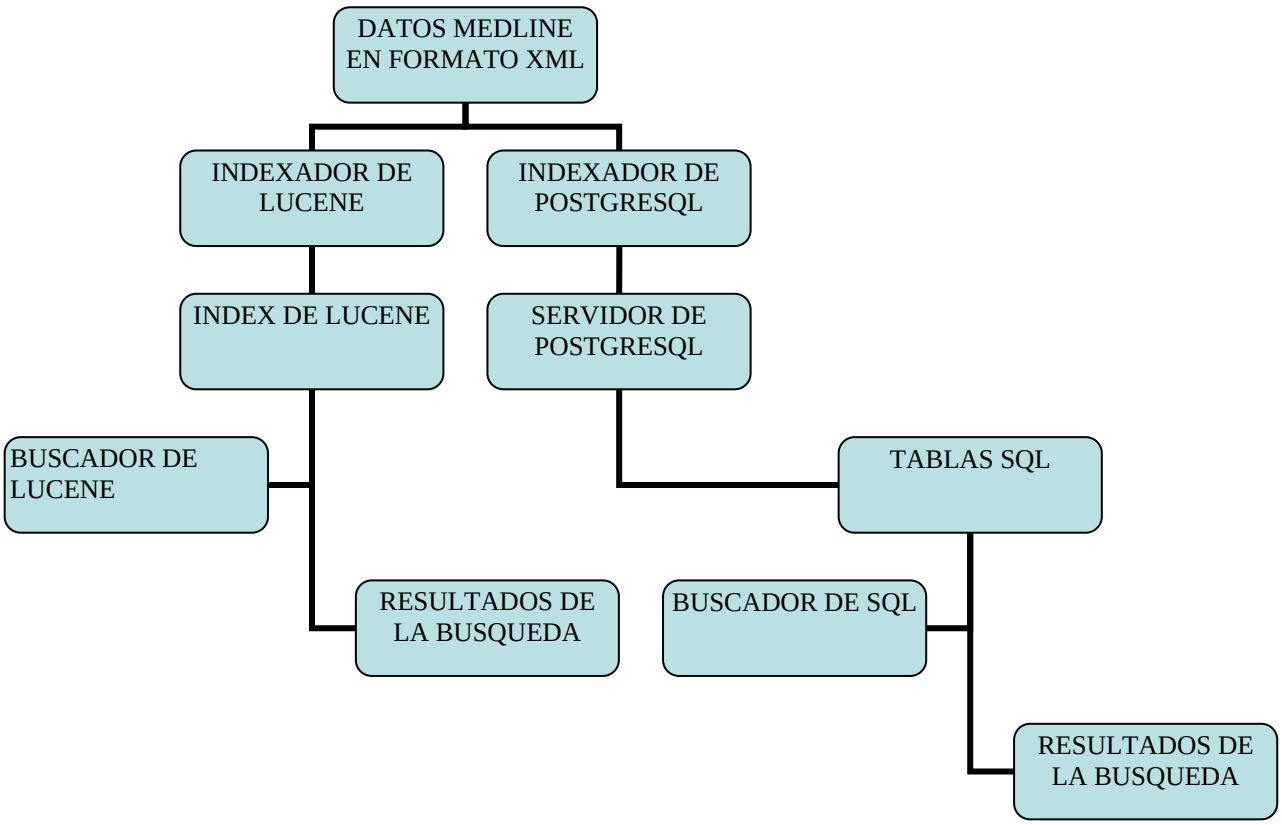
3) Permitir búsquedas relativamente complejas sobre los datos.

4) Evaluar el rendimiento de ambos sistemas.

5) Elegir cual es la mejor de las dos tecnologías.

1.4 Estructura y organización

A continuación se muestra un esquema aproximado de las relaciones de los distintos componentes del proyecto:



1.5 Estructura de los XML de MEDLINE

En este apartado explicaremos brevemente el significado de los campos más interesantes de los XML de MEDLINE:

-MedlineCitationSet: contiene el conjunto de los MedlineCitations

-MedlineCitations: Contiene toda la información sobre una referencia bibliografica.

-PMID: identificador único de la referencia.

-Article: contiene la información relativa al artículo al que hace referencia este MedlineCitation. Contiene campos como el titulo del artículo, sus autores, fecha de publicación, etc.

-Journal: Contiene información relativa a la revista que publico el articulo. Algunos de sus campos son nombre de la revista, autores, isbn, fecha de publicación, etc.

-MeshHeadingList: Contiene MeshHeadings que son términos designados por la NML, con el fin de caracterizar el contenido del articulo que representa esta MedlineCitation

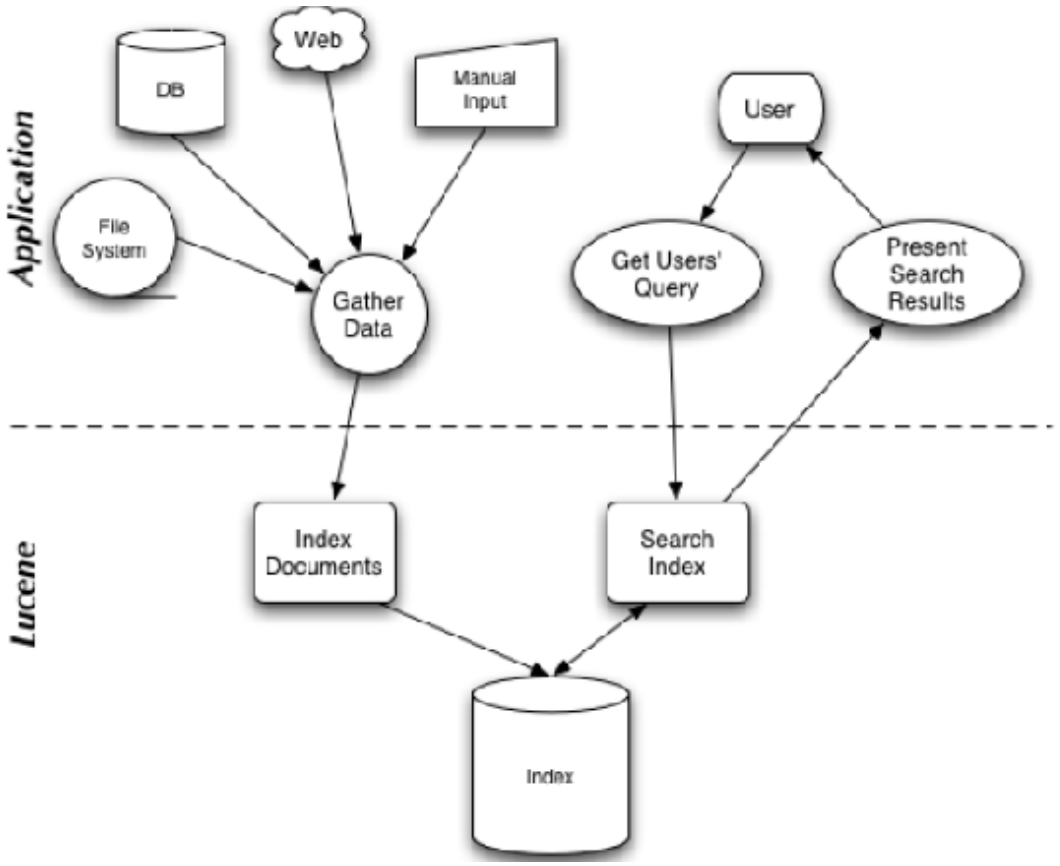
2. LUCENE

2.1 Visión general de Lucene

Lucene es una librería java de tratamiento de la información, que permite añadir indexación y búsqueda a las aplicaciones.

Lucene proporciona una simple API que permite indexar datos de tipo texto únicamente, con lo cual Lucene solo es adecuado con información que puede ser convertida a texto, pero resulta una herramienta bastante poderosa si salvamos este escollo.

Así el funcionamiento básico de Lucene es el que se muestra en la siguiente figura:



Para utilizar la API de Lucene solo es necesario conocer unas cuantas clases y métodos los cuales describiremos mas adelante.

La principal ventaja de Lucene frente a otras bases de datos es que ofrece a los usuarios capacidad para búsquedas totalmente textuales.

Aunque nosotros utilizamos la versión java de Lucene esta ha sido migrada a múltiples lenguajes de programación entre ellos C++, Perl, Phyton, .NET, etc.

En los siguientes apartados explicaremos las principales características y funciones de Lucene útiles para nuestro proyecto.

2.2 Indexando con Lucene

Para buscar grandes cantidades de texto rápidamente, es necesario indexar dicho texto y transformarlo a un formato que nos permita realizar búsquedas rápidamente, eliminando así el lento proceso de escanear los datos de entrada al completo cada vez que se realiza una búsqueda. Se puede pensar en un index como una estructura de datos que permite acceso aleatorio a las palabras que almacena. Estudiaremos mas detenidamente la estructura interna del index de Lucene mas adelante. El siguiente esquema muestra una visión general del proceso de indexación:

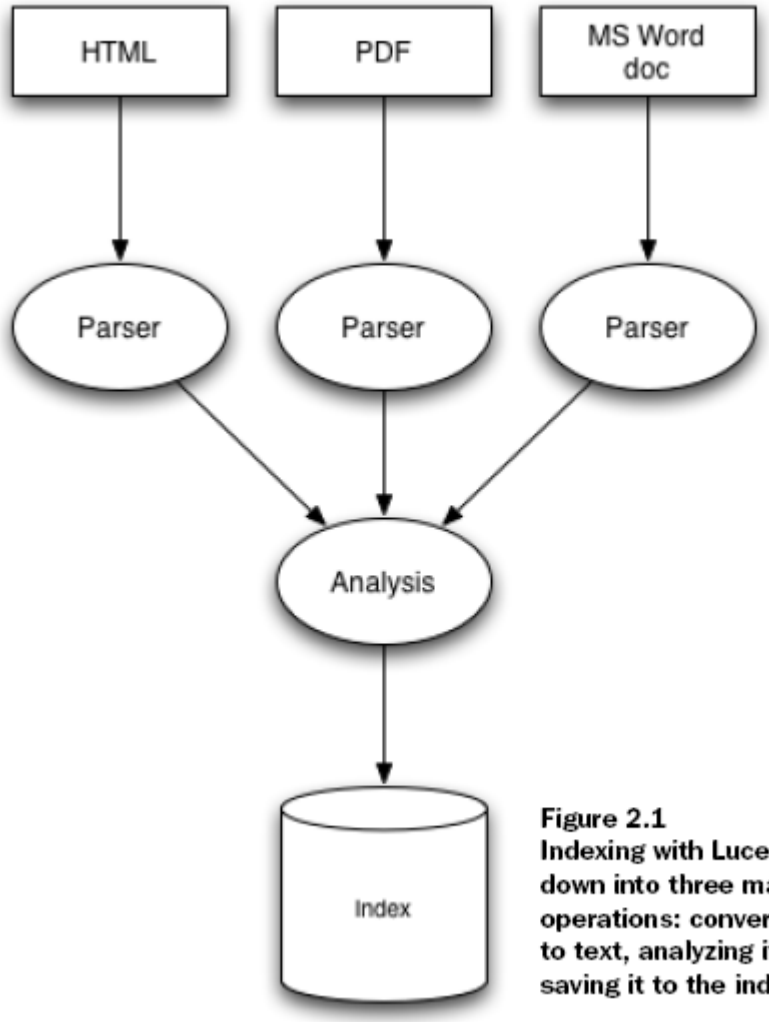


Figure 2.1
Indexing with Lucene breaks down into three main operations: converting data to text, analyzing it, and saving it to the index.

2.2.1 Clases para indexar del núcleo de Lucene

Para realizar un index es necesario conocer algunas clases:

- IndexWriter
- Analyzer
- Document
- Field

2.2.1.1 IndexWriter

Es el componente principal del proceso de indexación. Esta clase crea y añade documentos a un index, es decir, nos da permisos de escritura sobre un index pero no nos permite realizar búsquedas ni leer el index. La estructura de datos que conforma el index es también llamada index inverso, este tipo de estructura permite hacer eficiente el uso de espacio en disco y permite realizar búsquedas rápidas por palabras clave, ya que no trata a los documentos como el elemento central de la búsqueda, sino que busca por palabras claves.

2.2.1.2 Analyzer

Los datos antes de ser indexados deben pasar a través de un objeto de tipo Analyzer. El analyzer, el cual es especificado en la constructora de IndexWriter, es el encargado de extraer los tokens del texto que debe ser indexado y de eliminar el resto transformando la entrada en Objetos de tipo Term. El proceso de análisis se realiza cuando añadimos un documento al index con la función addDocument(Document) de IndexWriter, de manera que se analiza los datos para hacerlos mas manejables para indexar y, además de extraer los tokens puede realizarse sobre los datos una serie de operaciones opcionales como poner todo el texto en minúsculas para hacer las búsquedas no sensibles a mayúsculas o por ejemplo eliminar las típicas palabras que son irrelevantes a la hora de realizar búsquedas como artículos, preposiciones, etc , también tokenizar, lematizar, etc. Lucene ofrece muchos tipos diferentes de Analyzer según nuestras preferencias y operaciones que realizan a continuación veremos los más comunes:

WhitespaceAnalyzer: Divide los tokens en los espacios en blanco, no pone en minúsculas todo el texto y no elimina los guiones.

SimpleAnalyzer: divide el texto en los caracteres que no son letras y convierte todo a minúsculas.

StopAnalyzer: Hace lo mismo que el SimpleAnalyzer pero también elimina las stop words (palabras inservibles).

StandardAnalyzer: Tokeniza utilizando una sofisticada gramática que reconoce e-mails, acrónimos, caracteres orientales, alfanuméricos, etc. Pasa todo a minúsculas y elimina las stop words.

Existen muchos mas Analyzers específicos pero no los veremos, además Lucene nos permite implementar nuestro propio Analizar, implementando la interfaz Analyzer.
Si necesitamos para ciertos documentos usar otro Analyzer que el que hemos especificado en la constructora de IndexWriter es posible especificándolo en la función addDocument.

2.2.1.3 Document

Esta clase representa un conjunto de Fields, son los objetos utilizados para guardar la información en el index. Los Fields representan los metadatos de la información a indexar. El index de Lucene es heterogéneo, es decir, permite guardar en un mismo index documentos con diferentes fields, de manera que un único index puede almacenar documentos que representan entidades completamente distintas.

Boosting Documentos: Puede ser interesante darle más importancia a unos documentos que a otros a la hora de hacer búsquedas, para ello los documentos de Lucene tienen un parámetro que indica su importancia, el valor por defecto es 1.0, pero se puede modificar al crear el documento.

2.2.1.4 Field

Esta clase representa los metadatos de un Document, cada Document contiene uno o más Fields. Cada Field representa un fragmento de información sobre los que se harán preguntas durante el proceso de búsquedas en el index.
Lucene tiene 4 tipos diferentes de Fields para elegir, aunque en las últimas versiones de Lucene estas se han sustituido por un único Objeto Field personalizable para que sea igual que soporta estos 4 tipos:

Keyword: No es analizado pero si indexado y almacenado en el index. Este tipo es interesante cuando queremos preservar tal cual el valor original del texto que estamos indexando. Este tipo de Field puede recibir como valor una objeto de tipo Date.

DateField: Igual que Keyword pero solo puede recibir como valor un Objeto de tipo Date. Keyword internamente utiliza DateField para crear un String equivalente, pero se debe tener cuidado porque como Date contiene la fecha hasta los milisegundos puede provocar problemas de rendimiento para cierto tipo de querys. Otro problema es que este field no puede manejar

fechas anteriores al Unix Epoch (1 de enero de 1970). Por lo que suele ser preferible no usar esta opción.

UnIndexed: No es analizado ni indexado pero su valor es almacenado en el index tal cual. Este tipo es adecuado cuando necesitamos mostrar ciertos valores al realizar una búsqueda pero no podemos realizar búsquedas directamente sobre él, pero este tipo no es muy adecuado si el tamaño del index es importante y el field va a guardar valores muy grandes.

UnStored: Es analizado e indexado pero no guardado en el index. Es adecuado para guardar grandes cantidades de texto que no necesitan ser recuperados en su forma original nunca.

Text: Es analizado e indexado, por lo que se puede realizar búsquedas sobre estos fields. Si el tipo del texto indexado es un String también se guarda en el index pero si el tipo es Reader no es guardado.

Todos los fields son un par de valores, campo y valor, el elegir un tipo u otro depende de ara que se vayan a utilizar los datos.
También es posible no utilizar ninguno de estos 4 tipos y personalizar más los aspectos de nuestros fields.

Fields agregables: Una particularidad de Lucene, es que permite campos multivaluados, por ejemplo en nuestro proyecto necesitamos guardar para cada documento (referencia bibliografica de MEDLINE) los términos mesh los cuales tienen el mismo nombre y un numero variable de ellos por cada documento, una solución que proporciona Lucene es que añadimos varias veces el mismo campo con cada valor de los términos mesh, de manera interna Lucene agregara todas las palabras del mismo campo y las indexara en un mismo field de nombre el del campo, pero permitirá búsquedas para cada valor por separado.

Boosting fields: Al igual que los documentos se les puede asignar a los fields un valor que los haga más o menos importantes que el resto de fields del documento. El boost por defecto es 1.0 pero es modificable al crear el field.

Fields usados para ordenar resultados: Si queremos ordenar los resultados de una Query de acuerdo a un campo (por ejemplo, fecha) es

necesario añadir el campo como indexado pero no tokenizado, por ejemplo keyword, para que el valor sea convertible en integer, float y String, pero hay que tener cuidado porque Lucene utiliza orden Lexicográfico para ordenar.

2.3 Eliminar documentos de un index

La eliminación de documentos de un index se realiza con un objeto de la clase IndexReader. Esta clase no elimina los documentos directamente, sino que los marca como borrables y se espera para borrarlos a que se cierre el objeto IndexReader, debido a esta particularidad es posible revertir el proceso de borrado de documentos siempre que aun no se haya cerrado el objeto IndexReader.

2.4 Actualizar Documentos de un index

Lucene no ofrece un sistema para actualizar los documentos de un index, la única forma de hacerlo es borrar el documento que se quiere actualizar y reinsertarlo actualizado.

Si lo que queremos hacer es borrar y actualizar varios documentos la mejor opción siempre es borrar todos los que se quiere actualizar y volver a añadir todos y NO ir borrando uno y añadiéndolo y borrando otro y añadiéndolo, así sucesivamente, porque siempre es mucho mas rápido de esta forma.

2.5 Configuración del proceso de indexación

Lucene permite modificar ciertos aspectos del proceso de indexación para intentar hacer más rápido y eficiente dicho proceso, así podemos hacer que lucene aproveche la RAM disponible en la máquina que estamos utilizando para indexar. Normalmente el cuello de botella del proceso de indexación suele ser la escritura de los datos en disco. Los nuevos documentos añadidos al index no son directamente escritos en disco sino que son guardados en memoria, con el fin de mejorar el rendimiento la clase IndexWriter contiene varias variables públicas que permiten ajustar el tamaño del buffer de memoria y la frecuencia de escrituras en disco. Dichas variables son las siguientes:

mergeFactor

Permite controlar cuantos documentos son guardados en memoria antes de ser escritos en disco en un único segmento y también la frecuencia con la que se unen múltiples segmentos del index en un único segmento. Obviamente cuanto mayor es

este valor mayor es el uso de memoria RAM y menor la frecuencia de escritura en disco lo que provoca un aumento de la velocidad del proceso de indexación, pero reduce la frecuencia de unión de segmentos lo que provoca que el index tenga mas archivos, lo que hace mas lentas las búsquedas.

maxMergesDocs

Mientras se unen segmentos Lucene se asegura de que ningún segmento con más de maxMergesDocs documentos es creado. Su valor por defecto es Integer.MAX_VALUE.

minMergesDocs

Controla cuantos documentos deben ser almacenados en el buffer antes de unirse formando un segmento. Es decir, permite controlar la cantidad de RAM usada en el proceso de indexación, pero a diferencia de mergeFactor no afecta al tamaño de los segmentos en disco.

maxFieldLength

Para evitar que valores muy altos de los datos que contienen los fields sobrepasen la máxima memoria RAM utilizable se puede limitar estos campos a las primeras N palabras, tiene un valor por defecto de 10000. Este valor puede ser cambiado en cualquier momento no afectando a los documentos ya indexados pero si a los que se indexen en adelante.

Errores Habituales:

Hay que tener cuidado al modificar estos valores porque pueden desembocar en varios errores:

-OutOfMemory: Si ponemos valores de mergeFactor o minMergesFactor demasiado alto podemos sobrepasar el límite de memoria RAM que tenemos. A veces este problema se puede solucionar asignando mas memoria a la aplicación si tenemos memoria sin utilizar con las opciones –Xms y –Xmx de la máquina java.

-Too Many Files Open: Este error solo se da en UNIX y cuando usamos un multiFile index. Se puede intentar solucionar de varias maneras, como por ejemplo forzar una optimización del index con optimize() para que todos los segmentos actuales del index se unan en uno solo.

Si esta solución no funciona se puede incrementar el número de archivos abiertos permitidos:

% ulimit –n <máximo numero de archivos abiertos>

Para estimar el máximo número de archivos que Lucene tendrá abiertos en un momento dado usamos:

(1 + mergeFactor) * FilesporSegmento

Si el problema persiste es mejor cambiar el multiFiles index por un compound index que abre muchos menos archivos en todo momento, se darán más detalles sobre los tipos de index más adelante.

2.6 Optimizando el index

La optimización del index es el proceso de reducir el número de segmentos del index a uno para así minimizar el tiempo de búsqueda. Esta operación consume mucha entrada salida luego no puede ser utilizada a la ligera. Optimizar un index no afecta en absoluto al rendimiento del proceso de indexación, solo afecta a las búsquedas futuras sobre ese index, esto es así porque en el proceso de búsqueda Lucene solo deberá abrir y procesar menos archivos que en un index no optimizado. también es necesario tener en cuenta que durante el proceso de optimización no se borran los antiguos segmentos luego en el momento justo anterior a acabar la optimización el espacio utilizado por el index es el doble, al acabar de optimizar se borran los antiguos segmentos ocupando el nuevo la suma de todos los originales. También hay que tener en cuenta que las ID de los documentos puede cambiar al realizar una optimización.

2.7 Concurrencia

Debemos tener en cuenta las reglas de concurrencia así como las reglas cuando trabajamos con multi-threads al manejar un index.

2.7.1 Reglas de concurrencia

Existen varias reglas de concurrencia en lucene para evitar la corrupción del index:

-Cualquier número de operaciones de lectura simultánea están permitidas.

-Cualquier numero de operaciones de lectura simultanea están permitidas mientras se esta produciendo una operación de modificación.

-Solo una operación de modificación esta permitida simultáneamente. Así solo puede tener abierto el index un objeto IndexWriter o IndexReader a la vez.

2.7.2 Reglas de concurrencia en multi-threads

Es posible que varios threads trabajen con el index simultáneamente si utilizan el mismo objeto de IndexWriter o IndexReader, en estos casos Lucene debe asegurarse que no se solapan operación no permitidas simultáneamente para ello sigue las siguientes reglas:

-Un documento no puede ser añadido (IndexWriter) mientras un documento esta siendo borrado (IndexReader).

-Un documento no puede ser borrado (IndexReader) mientras el index está siendo optimizado (IndexWriter).

-Un documento no puede ser borrado (IndexReader) mientras el index esta uniendo segmentos (IndexWriter).

2.7.3 Uso de cerrojos para el index

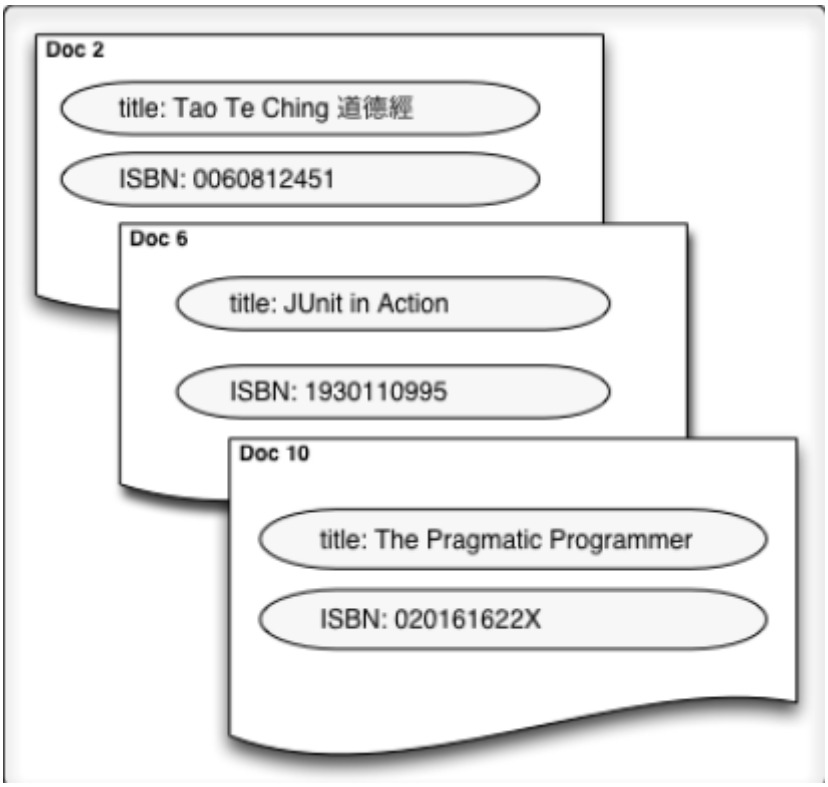
Para evitar la corrupción del index Lucene utiliza cerrojos. Al crear un Objeto de tipo IndexWriter o IndexReader Lucene crea un cerrojo (write.lock) en un directorio temporal que suele ser el mismo que donde se esta creando el index o cuando esta uniendo segmentos, IndexReader esta abriendo un segmento o IndexWriter esta creando un segmento nuevo se crea commit.lock pero este lock se libera nada mas acabar dichas operaciones con lo que no suele durar mucho tiempo a diferencia de write.lock. El directorio puede cambiarse dando valor a org.apache.lucene.lockDir. He aquí una tabla resumen del uso de estos cerrojos:

Lock File	Class	Obtained in	Released in	Description
write.lock	IndexWriter	Constructor	close()	Lock released when `IndexWriter` is closed
write.lock	IndexReader	delete(int)	close()	Lock released when `IndexReader` is closed
write.lock	IndexReader	undeleteAll(int)	close()	Lock released when `IndexReader` is closed
write.lock	IndexReader	setNorms (int, String, byte)	close()	Lock released when `IndexReader` is closed
commit.lock	IndexWriter	Constructor	Constructor	Lock released as soon as segment information is read or written
commit.lock	IndexWriter	addIndexes (IndexReader[])	addIndexes (IndexReader[])	Lock obtained while the new segment is written
commit.lock	IndexWriter	addIndexes (Directory[])	addIndexes (Directory[])	Lock obtained while the new segment is written
commit.lock	IndexWriter	mergeSegments (int)	mergeSegments (int)	Lock obtained while the new segment is written
commit.lock	SegmentReader	doClose()	doClose()	Lock obtained while the segment's file is written or rewritten
commit.lock	SegmentReader	undeleteAll()	undeleteAll()	Lock obtained while the segment's .del file is removed

Aunque esta totalmente desaconsejado desactivar los cerrojos de Lucene se puede hacer cambiando la propiedad del sistema disableLuceneLocks a “true”.

2.8 Formato del index de Lucene

Cuando indexamos creamos documentos que contienen fields, que son un nombre y un valor como muestra la siguiente figura:

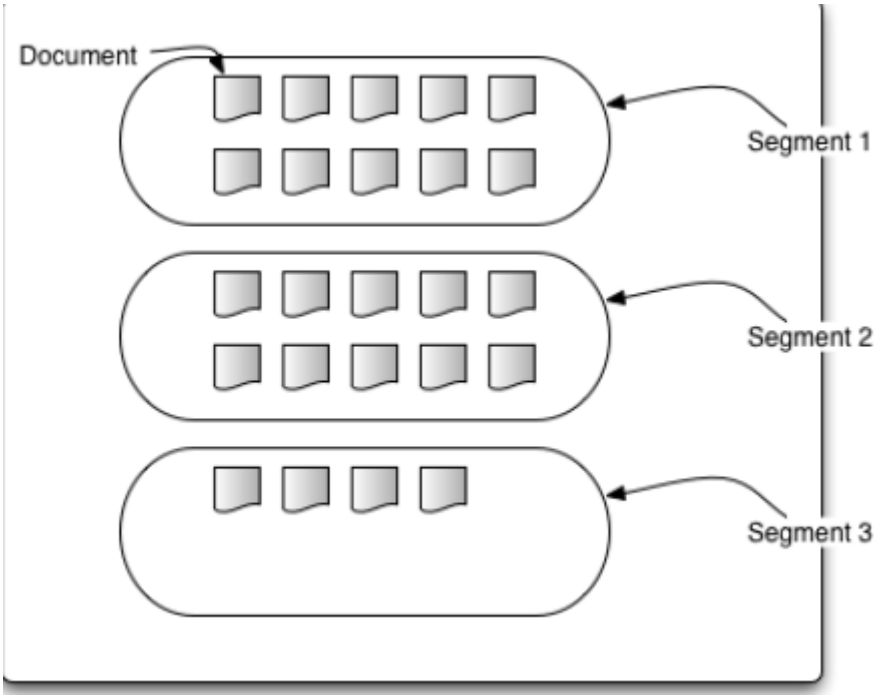


En este apartado veremos un poco cual es la estructura de un index, así como los dos tipos de index que hay.

Lucene soporta dos estructuras para su index, multifile index y compound index

Multifile index

Si observamos el directorio del index una vez creado observaremos varios archivos entre ellos varios comparten el mismo prefijo. Un segmento se compone de varios archivos del index, así aquellos archivos que comparten prefijo pertenecen a un mismo segmento. Cada segmento contiene varios documentos como se muestra en la siguiente figura:



El uso de segmentos permite añadir rápidamente nuevos documentos añadiéndolos a un segmento creado recientemente y uniéndolos periódicamente. Este proceso hace la agregación de nuevos documentos eficiente porque minimiza las modificaciones físicas del index.

El index de Lucene soporta indexación incremental es decir, tras añadir un documento este es inmediatamente accesible, mientras que en otros muchos sistemas primero se debe reindexar todo, esto hace que Lucene sea adecuado para tratar con grandes valores de texto.

Los archivos que componen el index contienen información que los relaciona entre ellos de manera que si se elimina un archivo accidentalmente el index se vuelve corrupto pero si al index le añadimos archivos al azar este no se corrompe.

El index también tiene un archivo segments que contiene el nombre de todos los segmentos contenidos en el index.

El número de archivos del index depende del número de fields indexados, el archivo del segmento que guarda esta información es el que tiene extensión fN con N un número natural.

Compound index

Es el index por defecto, a diferencia del multifiles index que debe abrir muchos archivos cuando accede al index, éste solo debe abrir dos.

Mientras que en el multifile cada segmento tiene 7 archivos, en el compound index cada segmento solo tiene un archivo .cfs que encapsula todos los index files.

En general, crear un multifiles index es entre un 5% y un 10 % mas rápido que crear un compound index, pero este solo es viable si la base de datos no es demasiado grande, puesto que sino tendremos muchísimos archivos en el index, con el consecuente riesgo de obtener el error “too many files open”.

2.9 Búsquedas con Lucene

En este apartado estudiaremos las posibilidades de búsqueda que nos ofrece Lucene, dejando las muy avanzadas de lado. Las búsquedas son uno de los puntos fuertes de Lucene siendo la recuperación de información del index muy rápida y permitiendo realizar gran variedad de consultas. La API de Lucene destinada a las búsquedas es muy sencilla y apenas requiere unas cuantas líneas de código, a continuación describiremos las principales clases necesarias para realizar las búsquedas.

2.9.1 Clases para búsqueda del núcleo de Lucene

Las clases destinadas a realizar búsquedas son las siguientes:

-IndexSearcher

-Term

-Query

-TermQuery

-Hits

-Filter

-QueryParser

2.9.1.1 IndexSearcher

Esta es una clase que accede al index y tiene permiso para leerlo, pero no puede modificarlo. Esta clase tiene un método search (Query) que es el encargado de realizar la búsqueda en el index y de devolvernos un objeto de tipo Hits con los resultados de la query.

2.9.1.2 Term

Es la unidad básica de búsqueda. Consiste en un par de Strings correspondientes al nombre de un field y al valor que contiene. Estos objetos también son utilizados internamente por Lucene al crear el index de ahí que se utilicen ahora para las búsquedas.

2.9.1.3 Query

Esta es una clase abstracta, Lucene viene con varias subclases de Query que implementan varios tipos de querys: TermQuery, BooleanQuery, PhraseQuery, PrefixQuery, PhrasePrefixQuery, WildCardQuery, FuzzyQuery, RangeQuery y SpanQuery.

TermQuery: Este es el tipo más básico de query, se usa para encontrar documentos con ciertos valores en campos concretos. Para construir la Query utiliza Terms que como ya sabemos es la unidad más pequeña de búsqueda. Como no utiliza Analyzer hay que tener en cuenta que es sensible a mayúsculas.

RangeQuery: Los términos en el index son ordenados lexicograficamente, permitiendo que las búsquedas por rango sean eficientes. Este tipo de query realiza búsquedas de documentos que tienen un valor entre dos especificados. Al crear la RangeQuery se puede especificar si están o no incluidos los términos de comienzo y fin de intervalo.

PrefixQuery: Se utiliza para buscar Documentos que contienen valores que comienzan por un prefijo dado en un field concreto.

BooleanQuery: Todas los tipos de querys pueden ser combinados utilizando una BooleanQuery, en otras palabras, este tipo de query es un contenedor de cláusulas booleanas o expresiones que pueden ser opcionales, obligatorias o prohibidas, lo que es equivalente a OR, AND y NOT. Estas querys tienen un número máximo de cláusulas el cual por defecto es 1024, pero este valor es modificable si fuera necesario.

PhraseQuery: El index contiene información de la posición de los términos. Este tipo de Query utiliza esta información para encontrar Documentos que tengan dos palabras separadas por cierto numero de posiciones, a la distancia máxima se la llama slop y Distance al numero de movimientos de terms necesarios para reconstruir la frase. Por defecto el slop es 0 pero obviamente se puede poner el que se quiera. Se puede construir una PhraseQuery con más de dos términos en cuyo caso el slop será el máximo valor de la suma de movimientos necesarios para colocar todos los términos en orden. El score de este tipo de querys, es decir, el orden que tendrán en el objeto Hits, esta directamente relacionado con los movimientos necesarios para ordenar la frase, así cuantos menos movimientos mejor score obtendrá el documento.

WildCardQuery: Permite realizar búsquedas de términos a los que le faltan caracteres, para ello utiliza dos operadores: el * que indica cero o más caracteres y ? para cero o un carácter. Si las palabras buscadas tienen grandes prefijos, estas querys pueden degradar seriamente el rendimiento, por ello es recomendable no comenzar los patrones con el símbolo *. Es por ello que si utilizamos QueryParser para hacer nuestras querys se restringe el uso de los símbolos de las wildcards para que no puedan ser utilizadas al principio.

FuzzyQuery: Realiza la búsqueda de términos similares al dado por la query. Para saber cuanto de similares son dos términos Lucene utiliza la distancia de Levenshtein que es medida como el numero de borrados, inserciones y sustituciones de caracteres requeridas para transformar un String en el otro. El verdadero valor que utilizara una FuzzyQuery es el siguiente:

1-**(**distance/min(textlen, targetlen))

Las FuzzyQuerys ordenan automáticamente los resultados de más parecido a menos.

PhrasePrefixQuery: Similar a PhraseQuery, solo que permite varias palabras diferentes por posición como validas. Es equivalente a utilizar una BooleanQuery con varias PhraseQuerys combinadas con OR. Al igual que las PhraseQuerys tienen slop. Esta query resulta útil cuando queremos que se acepten sinónimos para cierta posición.

SpanQuery: Existe toda una familia basada en SpanQuery que permite búsquedas muy sofisticadas relacionadas con la posición de las palabras en los campos de manera que se puedan buscar conjuntos de palabras próximos a otros por ejemplo.

2.9.1.4 Hits

Es el objeto devuelto tras la búsqueda y que contiene los resultados de la Query, en concreto contiene los documentos que satisfacían la Query. Por razones de rendimiento este objeto no guarda todos los documentos que satisfacen la Query sino que en cada momento solo guarda una porción de ellos, por defecto los 100 primeros. Los resultados vienen por defecto ordenados por el score, que es un número que calcula Lucene en base al documento de manera que es más alto cuanto más satisfaga el documento la Query aunque Lucene nos permite cambiar este orden.

2.9.1.5 Filter

Una opción muy sofisticada de personalizar aun más las querys consiste en utilizar filtros en las búsquedas de manera que se restrinja, por ejemplo, la query a un subconjunto del index. Existen 3 tipos de filtros:

DateFilter: Reduce los documentos sobre los que se realiza la búsqueda a un rango de fechas. DateFilter permite crear filtros con el rango abierto, es decir, antes o después de una fecha.

QueryFilter: Utiliza como espacio valido para realizar la query el resultado de otra query.

CachingWrapperFilter: Se utiliza para "guardar" los resultados de otro filtro de manera que se aumente el rendimiento al volver a usar el filtro.

2.9.1.6 QueryParser

Procesa una expresión dada por un usuario y la convierte en un objeto de la clase Query de los vistos anteriormente según corresponda. Este objeto requiere un Analyzer para transformar la expresión de entrada en Terms. Se pueden configurar algunas cosas de este objeto como por ejemplo cual es el operador por defecto en las expresiones que en principio es OR pero se puede cambiar. Veamos la sintaxis de las expresiones:

Sintaxis:

- Palabra→ Busca Documentos con el valor palabra en su field por defecto.
- Palabra1 palabra2→ Equivalente a palabra1 OR palabra2, busca documento que contienen una de las dos palabras o las dos en su field por defecto.
- +palabra1 +palabra2→ equivalente a palabra1 AND palabra2, busca documentos que tienen las dos palabras en su field por defecto.
- field:palabra → Busca documentos que contienen palabra en el field Field.
- Field:(palabra1 palabra2)→ Busca documentos que contienen las dos palabra en el field Field, no siendo necesario que estén juntas.
- -palabra→ equivalente a NOT palabra en el field por defecto, también se puede combinar con otro fields (ej. –field:palabra)
- Se puede utilizar en las expresiones AND, OR y NOT, en vez de lo descrito anteriormente.

- Cuando se utiliza NOT o – es necesario que haya al menos un término no negado, es decir no se pueden hacer querys que busquen todos los documentos que no contengan una palabra únicamente.
- Expresión –field:palabra → equivalente a expresión AND NOT field:palabra
- Se pueden utilizar paréntesis para alterar el la preferencia de los operadores o hacer subquerys.
- Field:"frase"→ Documentos que contienen exactamente la frase en el field Field
- Field:"palabra1 palabra2"~n→ contiene en field las palabras palabra1 y palabra2 separadas por n posiciones.
- Prefijo*→ contiene palabras que tienen el prefijo prefijo.
- ? palabra→ Documentos que contienen el termino palabra precedido por una o cero letras.
- Palabra~→ contiene palabras parecidas a palabra
- Field:[valor1 TO valor2]→ Documentos con valor en el field entre los dos valores ambos inclusive. Si utilizamos {valor1 To valor2} es ambos excluidos del intervalo. Los rangos se pueden utilizar con fechas si el field es de tipo Date.
- Para utilizar símbolos reservados en nuestras querys debemos precederlos de \.
- Palabra^float→ Asigna boost factor a la query precedente.

2.9.2 Ordenando los resultados de la búsqueda

Para ordenar los resultados a nuestro antojo es necesario utilizar para las búsquedas el método search(Query, Sort) de IndexSearcher en lugar de search(Query). Para ello Lucene trae por defecto varias formas de ordenar los resultados:

Sort.RELEVANCE: Equivalente a pasarle null o new Sort(). Es el orden por defecto de mayor relevancia a menor utilizando el score de Lucene, es decir, de los que más encajan con la query a los que menos.

Sort.INDEXORDER: Ordena los documentos por el orden en el que fueron introducidos en el index.

Ordenando de acuerdo a un field: Para ordenar de acuerdo a un field este debe haber sido creado de acuerdo a lo que dijimos en el apartado **fields usados para**

ordenar resultados. Para ordenar de esta forma debemos crear un objeto de la clase Sort pasándole a su constructora el nombre del field. El orden por defecto es incremental pero se puede cambiar indicándolo en la constructora de Sort.

Ordenando de acuerdo a varios fields: Para ordenar de esta manera le pasamos a la constructora un array de SortField de manera que se ordenara de acuerdo al primero y en caso de empate de acuerdo al segundo y así sucesivamente.

3. POSTGRESQL

3.1 Visión general de PostgreSQL

PostgreSQL es una de los sistemas de bases de datos relacionales más potentes que existen, estos sistemas permiten guardar datos en entidades bidimensionales llamadas tablas.

Nació en 1977 como un proyecto llamado Ingres en la universidad de California que más tarde se comercializaría y en 1986 se ampliaría dicho proyecto y se le cambiaría de nombre al actual PostgreSQL. Una de las principales cualidades de PostgreSQL es que a pesar de ser código abierto ofrece muchas cualidades normalmente exclusivas de los sistemas comerciales como son fácil extensibilidad, integridad referencial, lenguajes procedimentales, control de concurrencia multi-versión, etc.

PostgreSQL utiliza el paradigma de servidor-cliente, así el servidor es el encargado de realizar todas las tareas, se ejecuta en background y escucha a través de un puerto las peticiones de los clientes y las realiza si es posible.

Hemos elegido PostgreSQL para indexar MEDLINE de entre todas las bases de datos relacionales porque es la más robusta frente a bases de datos muy grandes y además como hemos visto ofrece gran variedad de opciones.

En los próximos apartados veremos la sintaxis básica utilizada por PostgreSQL y que puede ser útil en nuestro proyecto tanto ahora como para futuras ampliaciones así como las principales características de este sistema sin excedernos demasiado en detalles puesto que este tipo de sistemas son ya muy conocidos y no parece necesario.

3.2 Conceptos básicos

Tablas

Las tablas representan relaciones entre la información de la base de datos, están compuestas por filas y columnas y la intersección de estas campos, los campos son los encargados de contener la información, la cual puede ser de tipos muy variados como enteros, floats, String, etc. Las filas representan el conjunto de valores correspondientes a un objeto concreto y las columnas representan los campos y el tipo de cada componente de las filas. PostgreSQL define para cada tabla una serie

de columnas del sistema, las cuales normalmente no son visibles al usuario. Estas columnas contienen metadatos relativos a las tablas. Estas columnas son las siguientes:

-oid: identificador único para cada fila de la tabla

-tableoid: Identificador único de la tabla que contiene la fila.

-xmin: Identificador de la transacción que insertó los datos de la tupla.

-cmin: Identificador único del comando que insertó los datos de la tupla en la tabla.

-xmax: Identificador de la transacción que borró los datos de la tupla, si aun no han sido borrados este valor es 0.

-cmax: Identificador del comando que borró los datos de la tupla, si aun no han sido borrados este valor es 0.

-ctid: identificador de la localización física de la tupla. Son dos valores el número del bloque y el index de la tupla en el bloque.

Token

Los tokens son las unidades básicas de las sentencias SQL, pueden ser de diversos tipos: Keywords, identificadores, identificadores entrecomillados, constantes o símbolos de caracteres especiales.

Es importante saber que todos los identificadores no entrecomillados son convertidos a minúsculas.

Por tanto, solo es útil entrecomillar los identificadores cuando queremos tener en cuenta mayúsculas o algún objeto de nuestra base de datos relacional tiene el mismo nombre que una palabra reservada.

Los identificadores y las palabras reservadas pueden tener un número máximo de caracteres de 31, todos aquellos que sobrepasen este límite son automáticamente truncados.

Operadores

Es un tipo de carácter especial que representa una operación a realizar entre identificadores o constantes devolviendo valores. PostgreSQL ofrece varios operadores por defecto de sobras conocidos como:

-Operadores matemáticos: +, -, /, *, !, @ (valor absoluto), %, ^, |/ (raíz cuadrada), ||/ (raíz cúbica), !!.

-Operadores de comparación: =, <, >,<=, >=, <> ó !=, between, is null.
-Operadores Lógicos: and, or, not.
-Comentarios: --, /* */.
-Operadores para subquerys: IN, EXISTS
-Operadores binarios: &, |, # (XOR), ~(not binario), <<, >>.
También están los operadores para texto pero veremos estos más en detalle más adelante.

Vistas

Las vistas representan el conjunto de valores devueltos por una query. Por ello son como una tabla dinámica cuyos valores se calculan en tiempo de ejecución. Las vistas una vez creadas pueden ser utilizadas en cualquier sitio donde se utilizaría una tabla excepto sentencias de modificación de datos, puesto que los datos no pertenecen realmente a la vista sino a las tablas que usaba la query que se utilizó para crear la vista. Normalmente se utilizan para sustituir a querys informativas.

Funciones

Una función es un identificador que realiza una función determinada, puede recibir argumentos y devuelve un valor determinado, estas funciones solo son utilizadas dentro de una sentencia SQL y no tienen nada que ver con las funciones de agregación (las cuales veremos más adelante, cuando veamos la sintaxis de las consultas). Existe gran variedad de funciones definidas en PostgreSQL:
-Funciones matemáticas: abs(x), acos(x), asin(x), atan(x), atan2(x,y), cbrt(x), ceil(x), cos(x), cot(x), negrees(x), exp(x), floor(x), ln(x), log(b,x), log(x), mod(x,y), pi(), pow (x,y), radians(d), random(), round(x), round(x,s), sin(x), sqrt(x), tan(x), trunc(x), trunc(x,s).
-Funciones para caracteres y strings: ascii(s), btrim(s[, t]), char_length(s), chr(n) s ilike(f), initcap(s), length(s), s like(f), lower(s), lpad(s, n[, c]), ltrim(s[, t]), octet_length(s), position(b IN s), repeat(s,n), rpad(s, n[, c]), rtrim(s[, t]), strpos(r,b), substr(s, n[,l]), substring(s FROM n FOR l), to_ascii(s,f), translate(s,f,r), trim(SIDE f FROM s), upper(s).
-Funciones para fechas y horas: current_date, current_time, current_timestamp, date_part(s,t), date_part(s,i), date_trunc(s,t), extract(k FROM t), extract(k FROM i), isfinite(t), isfinite(i), now(), timeofday().

La variable i corresponde a un interval y t a un timestamp.
-Funciones para conversiones de tipos: bitfromint4(n), bittoint4(n), to_char(n,f), to_date(s,f), to_number(s,f), to_timestamp(s,f), timestamp(d), timestamp(d,t)

Postgresql también nos permite crear nuestras propias funciones para posterior uso en nuestra base de datos.

Índices

Los índices son objetos de las bases de datos que permiten mejorar el rendimiento de la misma al mejorar la velocidad de ejecución de sentencias que involucran cláusulas condicionales de acceso a valores de una columna. Pero hay que tener cuidado al crear índices, porque necesitan un mantenimiento y hacen más lento el proceso de inserción de datos, por ello los índices solo deberían ser usados en columnas que cumplan ciertos criterios.

-Criterios:

1. Se accede mucho a esa columna en cláusulas WHERE.
2. Tiene pocos valores repetidos.
3. Que se seleccionen pocas filas cuando se utiliza el índice.
4. Que se realicen pocas operaciones update sobre los datos.

Existen varios tipos de índices en PostgreSQL.

-Tipos de índices:

Índice UNIQUE→ Impide que existan valores repetidos en la columna o tuplas de valores de las columnas que se usan para el index. Aunque como NULL no es considerado un valor este si puede estar varias veces.

Índice B-Tree→ Utiliza los algoritmos de Lehman-Yao para alta concurrencia. Es el más utilizado y el tipo por defecto. La implementación consiste en un árbol binario ordenado de los valores de la columna.

Índices R-Tree→ Es bastante útil cuando se realizan operaciones espaciales sobre los datos (por ejemplo operaciones geométricas). Utiliza el algoritmo quadratic split de Guttman.

Índices Hash→La implementación consiste en una tabla hash. Utiliza las rutinas lineales de Litwin para hash. Este tipo de índices es bastante útil cuando se realiza frecuentemente operaciones de comparación de igualdad en los datos.

Aunque este tipo de index esta un poco obsoleto y en general funciona mejor el B-Tree.
Se pueden crear índices multi-columnas o compuesto, pero estos solo serán utilizados cuando se acceda a todas las columnas del index en cláusulas AND de WHERE. El número de columnas máximo por índice es de 16 y, para este caso, solo se puede utilizar el tipo de índice B-tree.
El uso de PRIMARY KEY crea implícitamente un índice sobre las columnas que conforman la key.
Existe en PostgreSQL una modificación del índice tradicional y es que se permite crear un índice sobre valores de funciones aplicadas a columnas en vez de a los valores de las columnas directamente.

Triggers

Se utilizan los triggers para asociar acciones a ciertos eventos sobre ciertos datos particulares. Estas acciones se pueden incluir en el código que maneja la base de datos, pero si se crea un trigger también se puede hacer de una manera más automática. Se suelen usar los triggers para verificar que los datos cumplen ciertas condiciones antes de ser insertados, ó para propagar borrados o actualizaciones de los datos. La acción del trigger se puede definir para que ocurra inmediatamente antes o inmediatamente después del evento que la desencadena. El cuerpo del trigger se implementa en cualquier lenguaje excepto SQL que se pueda utilizar para definir funciones en PostgreSQL. Los eventos que pueden desencadenar un trigger son INSERT, UPDATE y DELETE.

3.3 Tipos

La información que se guarda en postgreSQL puede ser de diversos tipos.

3.3.1 Constantes

El parser de PostgreSQL reconoce 5 tipos diferentes para las constantes introducidas:

String: Es una secuencia de caracteres entre comillas simples. Existen una serie de caracteres especiales que se pueden incluir en un String: \\, \', \b, \f, \n, \r, \t, *octal_number*. Si en una sentencia se escriben dos Strings consecutivos separados por un salto de línea estos se concatenan automáticamente.

Bit String: Permite representar cadenas binarias. Se representan entre comillas simples inmediatamente precedido de la letra b ó B. Al igual que los Strings se pueden representar utilizando varias líneas concatenándose el valor automáticamente.

Integer: Cualquier secuencia de números sin parte decimal y fuera de comillas simples es interpretada como valor entero. PostgreSQL utiliza por defecto los enteros con signo de 4 bytes, por tanto su rango de valores es -2147483648 hasta 2147483647.

Floating Point: Es cualquier secuencia de números con o sin parte decimal y fuera de comillas. Su representación es la habitual de punto flotante es decir valores como 12.5, 1.2 –e24, etc.

Boolean: Se representan fuera de comillas, true o false, aunque internamente PostgreSQL los representa como t y f si ponemos tal cual estos valores también serán reconocidos como booleanos al igual que 1/0, yes/no, etc.

3.3.2 Tipos de las columnas

Los tipos permitidos para definir las columnas son mucho más numerosos por lo que solo los enumeraremos:

Tipos Booleanos y Binarios: boolean ó bool, bit(n), bit varying(n) ó varbit(n).

Tipos Caracteres: carácter(n) ó char(n), carácter varying(n) ó varchar(n), text.

Tipos Numéricos: smallint ó int2, integer ó int ó int4, bigint ó int8, real ó float4, double precisión ó float8 ó float, numeric(p,s) ó decimal(p,s), Money, serial.

Tipos de fechas y horas: date, time, timewithtimezone, timestamp, interval.

Tipos geométricos: box, line, lseg, circle, path, point, polygon.

NetWork Types: cidr, inet, macaddr.

System types: oid, xid.

3.3.3 Array

Los arrays son un tipo especial que incluye PostgreSQL. En principio las celdas de las tablas solo contienen valores atómicos pero gracias a los arrays es posible hacer que contengan un conjunto de valores. Los array son un conjunto de valores referenciados a través de un identificador. Todos los valores del array deben ser del

mismo tipo pero este puede ser cualquiera de los que distingue PostgreSQL o uno construido por el usuario.
Se permite a los arrays ser multi-dimensionales y también de longitud variable o fija.

Operadores de los array

nombre_array[n]→accede a la posición n del array.
'{value1, value2,…..}'→ Constante que representa un array.

3.4 Sintaxis básica de PostgreSQL

Structured Query Language (SQL) es un potente y versátil lenguaje para bases de datos relacionales creado por IBM y que ha evolucionado muchísimo hasta nuestros días convirtiéndose en el lenguaje estándar de las bases de datos relacionales.
PostgreSQL al recibir una sentencia, internamente, la divide en tokens por los espacios en blanco (o en algunos casos por comillas dobles, simples, etc.), estos tokens son interpretados por el servidor el cual es el encargado de completar la sentencia.
A continuación veremos la forma más básica de combinar tokens para formar sentencias SQL validas.

Crear una base de datos

```
CREATE DATABASE dbname
    [ WITH [ LOCATION = 'dbpath' ]
        [ TEMPLATE = template ]
        [ ENCODING = encoding ] ]
```

Crea una base de datos vacía de nombre el indicado. La opción template hará que se copien todos los objetos creados para esta base de datos en el directorio especificado, por defecto se utiliza el template1. El encoding hace referencia a con que criterio se codificarán los datos de la base de datos, existen muchos tipos de encoding los principales y más interesantes tipos de encoding que ofrece PostgreSQL son:

SQL_ASCII→ Que utiliza el convenio de ASCII.
UNICODE→ Igual que el UTF-8

LATIN9→Novena versión del encoding de LATIN y que incluye el ingles y algunos lenguajes europeos. Utiliza el estándar ISO8859. Este es el encoding por defecto de PostgreSQL.

Crear tablas

CREATE [TEMPORARY | TEMP] TABLE table_name (

{ column_name type [column_constraint [...]] | table_constraint }

[, ...]

) [INHERITS (inherited_table [, ...])]

[AS *query]*

Si se crea la tabla como temporal esta será destruida al final de la sesión, además si tiene el mismo nombre que una existente al intentar acceder a ésta se accederá a la temporal.

En la sección inherits se puede definir de qué tablas hereda la que estamos creando.

La sección AS permite que al crear la tabla se inserten en ella los valores resultado de la query, para ello los tipos y el número de columnas deben coincidir con los de la tabla creada. Pero si utilizamos esta sección para crear una tabla no podemos poner restricciones, solo se podrán añadir posteriormente aquellas restricciones que permita el comando ALTER TABLE.

También se puede definir una serie de restricciones para las columnas o para la tabla completa. Las principales restricciones son las siguientes:

-<u>Restricciones para las columnas</u>:

UNIQUE→ Obliga a que no se repitan valores de esta columna en la tabla

PRIMARY KEY→ Hace que el atributo sea considerado la clave primaria de de la tabla es decir crea un index sobre esta columna, no permite repeticiones de valores y no permite que tome el valor NULL.

NOT NULL→ No permite el valor NULL para esta columna.

REFERENCES *Table_name(column_name)*→ Obliga a que los valores introducidos en esta columna existan en la columna de la tabla especificada previamente. Se pueden añadir funcionalidades adicionales a esta restricción como MATCH FULL| PARCIAL que nos indica si se permiten valores NULL, ó ON DELETE|UPDATE *action* utilizada para que cuando se haga una de estas operaciones en la columna referenciada se haga dicha acción, normalmente se usa con CASACADE para que así se borre en todas las tablas el dato.

DEFAULT *value*→ Asigna un valor por defecto a la columna.
CHECK (*condición*)→ Los valores introducidos en esta columna deben cumplir estas condiciones.

-Restricciones para las tablas:

PRIMARY KEY (column_name1, ….)→ Define la clave primaria de la tabla, que puede estar formada por varias columnas y al igual que en la restricción para columnas crea un index compuesto sobre estas columnas, no permite repeticiones de valores en las tuplas y no permite que tome el valor NULL ninguna de estas columnas.
FOREIGN KEY (*column_name1,…*) REFERENCES *Table_name(column_name1, …)*→ Crea una clave ajena es decir obliga a que los valores de las columnas especificadas existan en otras columnas de otra tabla concretos. Las foreign keys tienen las mismas opciones adicionales que REFERENCES en las columnas.
CHECK (*condición*)→ Los valores introducidos en la tabla deben cumplir estas condiciones.
UNIQUE (*column1, column2,…*)→ Obliga a que no se puedan repetir valores en las tuplas que forman las columnas.

Modificar tablas

-Añadir una columna a la tabla:

```
ALTER TABLE table
    ADD [ COLUMN ] column_name column_type
```

-Poner o quitar valores por defecto:

```
ALTER TABLE table
    ALTER [ COLUMN ] column_name
    { SET DEFAULT value | DROP DEFAULT }
```

-Renombrar una tabla:

```
ALTER TABLE table
    RENAME TO new_table
```

-Renombrar Columnas:

```
ALTER TABLE table
    RENAME [ COLUMN ] column_name TO new_column_name
```

-Añadir restricciones:

Solo se puede añadir restricciones de tipo foreign key y check una vez creada la tabla.

```
ALTER TABLE table
    ADD [ CONSTRAINT name ]
    { CHECK ( condition ) |
     FOREIGN KEY ( column [, ... ] )
            REFERENCES table [ ( column [, ... ] ) ]
            [ MATCH FULL | MATCH PARTIAL ]
            [ ON DELETE action ]
            [ ON UPDATE action ]
            [ DEFERRABLE | NOT DEFERRABLE ]
            [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
    }
```

-Cambiar dueño:

```
ALTER TABLE table
    OWNER TO new_owner
```

Insertar valores en una tabla

```
INSERT INTO table_name
    [ ( column_name [, ...] ) ]
    VALUES ( value [, ...] )
```

Añade valores nuevos a las columnas especificadas de la tabla. Pero en vez de añadir valores nuevos podemos añadir valores existentes en otra tabla utilizando

```
INSERT INTO table_name
    [ ( column_name [, ...] ) ]
    query
```

pero los valores retornados por al query deben ser del mismo tipo y numero de ellos que las columnas donde se quieren insertar los valores.

También se pueden insertar datos en una tabla a partir de los resultados de una query, esta tabla será implícitamente creada de la manera más simple posible:

```
SELECT select_targets
    INTO [ TABLE ] new_table
    FROM old_table;
```

La tabla especificada en INTO no debe existir de lo contrario obtendremos un error.

Realizar una consulta

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    target [ AS name ] [, ...]
    [ FROM source [, ...] ]
        [ [ NATURAL ] join_type source
        [ ON condition | USING ( column_list ) ] ]
        [, ...]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] sub-query ]
    [ ORDER BY expression
        [ ASC | DESC | USING operator ]
        [, ...] ]
    [ FOR UPDATE [ OF table [, ...] ] ]
    [ LIMIT { count | ALL } [ { OFFSET | , } start ] ]
```

Siendo source:

```
[ ONLY ] table [ [ AS ] alias [ ( column_alias [, ...] ) ] ] |
    ( query ) [ AS ] alias [ ( column_alias [, ...] ) ]
```

Y siendo target:

Nombres de columnas, funciones, expresiones, etc, que se quieren consultar, pudiendo ser renombrados dinámicamente con AS. Si se utiliza como target * entonces se considera como target todas las columnas de las fuentes menos las columnas del sistema.

La palabra reservada DISTINCT obliga a que los valores de las columnas o expresiones especificados no se repitan nunca.

Las fuentes normalmente están separados por comas lo que implica que se hace cross join para combinar las fuentes pero se puede modificar para que se unan de otra manera o bajo ciertas condiciones, pero también se puede utilizar como fuente una subquery, aunque no tenga mucha utilidad. Los tipos de join permitidos son:

Cross join→ Producto cartesiano

Inner join→ Producto cartesiano con condiciones obligatorias.

Outer join→ Igual que inner join excepto que obliga a que al menos una instancia de cada fila original aparezca en el resultado final. Existen tres tipos de estas uniones left, rigth y full para indicar de qué tabla queremos que aparezcan todas las instancias de las filas.

Si se añade la palabra reservada NATURAL solo se unirán las filas por valores iguales en columnas con el mismo nombre, USING es equivalente a NATURAL solo que permite especificar sobre que columna queremos que lo haga.

En la cláusula WHERE se especifican las condiciones, mediante una expresión, que deben cumplir los datos para ser seleccionados por la query, aquí si es interesante el uso de subquerys, pero se debe tener en cuenta que el resultado de la cláusula WHERE debe ser un valor booleano.

La cláusula GROUP BY …HAVING crea agregaciones de filas uniéndolas por valores iguales en las columnas especificadas y en el orden especificado. Solo las columnas utilizadas para crear la agregación serán seleccionables en el target salvo que estén siendo utilizadas en una función puesto que estas columnas pueden ser conjuntos de valores. Las funciones de agregación son:

-count
-max
-min
-sum
-avg
-stddev
-variance

Estas funciones no pueden ser utilizadas en la cláusula WHERE pero si en la HAVING que funciona igual que la cláusula WHERE pero solo puede incluir condiciones que utilizan estas funciones, no nombres de columnas sin más.

Con la cláusula UNION | INTERSECT | EXCEPT se puede combinar los resultados de la query con los resultados de una segunda query.

ORDER BY ordena los resultados de la manera especificada en la cláusula.

La cláusula FOR UPDATE activa el control de concurrencia la las filas seleccionadas de manera que estas no pueden ser modificadas hasta que se ejecute un commit.

Por ultimo, la cláusula LIMIT se utiliza para indicar cuantas filas se quieren seleccionar como máximo y a partir de cual (OFFSET).

-Expresiones CASE

PostgreSQL permite la utilización de una estructura CASE en el target de manera que es posible mostrar unos resultados u otros en función del criterio que queramos.

```
CASE WHEN condition1 THEN result1
     WHEN condition2 THEN result2
     [ ... ]
     [ ELSE default_result ]
 END [ AS alias ]
```

En “result” es posible tener una subquery pero ésta deberá solo retornar un valor sino obtendremos un error.

Modificar datos

Esta sentencia se utiliza para modificar los datos de filas ya existentes en tablas.

```
UPDATE [ ONLY ] table SET
       column = expression [, ...]
       [ FROM source ]
       [ WHERE condition ]
```

ONLY solo es interesante si la tabla tiene tablas hijas y no queremos que estas sean modificadas.

La cláusula FROM permite utilizar valores de otras tablas en las expresiones.

Borrar datos

Esta sentencia borra filas existentes en una tabla. El borrado es permanente salvo que no se este utilizando el auto-commit.

```
DELETE FROM [ ONLY ] table
       [ WHERE condition ]
```

ONLY al igual que en la sentencia update solo se utiliza cuando la tabla tiene tablas hijas y no queremos que sean modificadas.

Subquerys

Las subquerys son un recurso muy utilizado sobre todo en cláusulas WHERE, y tienen sus propias operaciones, las subquerys siempre se deben colocar entre paréntesis. A continuación veremos el uso de operadores con subquerys:

-Igualdad:

column_name = (subquery)

La subquery debe retornar un único valor de lo contrario obtendremos un error.

-IN:

(column_name1,…..) IN (subquery)

La expresión devolverá true si el conjunto de valores a la izquierda de la expresión se encuentra en alguna tupla a la derecha.

-EXISTS:

EXISTS (subquery)

La expresión devolverá true si la subquery retorna al menos una fila de valores. Se puede obtener el operador contrario con NOT EXISTS.

Crear vistas

Para crear vistas utilizamos la sentencia:

```
CREATE VIEW view
        AS query
```

Crear un índice

```
CREATE [ UNIQUE ] INDEX indexname ON table
      [ USING indextype ] ( column [ opclass ] [, ...] )
```

Crear un índice en la tabla indicada usando la columna indicada.

Donde indextype es la implementación del índice que queremos utilizar para crear este índice y opclass para indicar que operador debe ser utilizado en el ordenamiento de los valores.

También se puede crear un índice sobre varias columnas.

Crear un trigger

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR event ... ] }
            ON tablename
            FOR EACH { ROW | STATEMENT }
```

EXECUTE PROCEDURE functionname (arguments)

La opción EACH ROW|STATEMENT sirve para especificar si queremos que la acción se aplique a cada fila afectada por el evento o solo una vez por cada evento.

Crear una función

CREATE FUNCTION name ([argumenttype [, ...]])

RETURNS returntype

AS 'definition'

LANGUAGE 'languagename'

[WITH (attribute [, ...])]

Esta sentencia permite crear funciones nuevas que serán utilizables en la base de datos como cualquier otra función.

3.5 Características avanzadas

3.5.1 Expresiones regulares y operadores para texto

Operadores para texto

PostgreSQL ofrece varios operadores para texto compatibles con los tipos char, varchar y text. También tiene un método para representar expresiones regulares a fin de utilizarlas en los operadores para que encajen con un conjunto de valores.
Los operadores de comparación <, >, >=, <=, = y <> ó != son validos para comparar cadenas de texto. Para mayor y menor se utiliza el orden alfabético para ello utiliza el valor ASCII de los caracteres.
El operador de concatenación de PostgreSQL es ||, otros operadores útiles son LIKE e ILIKE que comparan dos cadenas de caracteres y miden su grado de similitud.

Operadores para expresiones regulares

Los operadores para las expresiones regulares comparan un valor textual con una expresión regular. A continuación vemos los operadores que existen:

~→ Devuelve true si el valor textual encaja con la expresión regular.

!~→ Operador contrario a ~.

~*→ Igual que el operador ~ salvo que es insensible a mayúsculas.

!~*→ Operador contrario a ~*.

Construcción de una expresión regular

Las expresiones regulares son valores textuales con caracteres especiales que le dan cierto significado extra. Dichos caracteres y su función son los siguientes:

^ expresión→ Encaja con los valores textuales que empiezan por expresión.

expresión$ → Encaja con los valores textuales que acaban por expresión.

. → Encaja con cualquier carácter.

[char1 char2…]→ Encaja con cualquier carácter que se encuentre entre las llaves.

[^char1 char2…]→Encaja con cualquier carácter que no se encuentre entre las llaves.

[char1-char2]→ Encaja con cualquier carácter que se encuentre en el rango delimitado por char1 y char2.

[^char1-char2]→ Encaja con el contrario al anterior.

char? →Encaja con cero o una instancia del carácter char.

char* →Encaja con cero o más instancias del carácter char.

char+→Encaja con una o más instancias del carácter.

expresion1|expresion2→ Encaja con valores textuales que encajan con expresión1 o con expresión2.

Se pueden utilizar los paréntesis para modificar la precedencia de estos caracteres.

Cualquiera de estos caracteres especiales precedidos por \\ hace que se tenga en cuenta el carácter como valor y no como carácter especial.

3.5.2 Transacciones

Se considera una transacción a la acción de interactuar de un conjunto de sentencias SQL con la base de datos. Esto no implica que se haya aplicado físicamente la sentencia a la base de datos, esto no ocurrirá hasta que se haga un commit.

Postgresql utiliza un control de concurrencia multi-versión, que permite a las sentencias SQL ser ejecutadas como transacciones de bloques diferidas. Esto quiere decir que cada conexión a la base de datos mantiene una copia temporal de la base de datos sobre la que se realizan las modificaciones hasta que estas son commited, y por tanto modificaciones en la base de datos temporal no serán visibles para otros usuarios conectados a la base de datos.

Por defecto PostgreSQL utiliza el autocommit, lo que hace que no haya copias temporales de la base de datos.
Cuando se realizan cambios sobre una base de datos temporal estos pueden ser revertidos hasta la última vez que se ejecutó un commit con rollback.
Para comenzar una transacción de bloques se utiliza *begin;*

4. JDBC

4.1 Visión general

JDBC es una API programada en java que nos permite trabajar con bases de datos relacionales desde programas implementados en lenguaje java.

Para el proyecto hemos utilizado la versión específica para PostgreSQL, aunque las partes utilizadas en el proyecto están en cualquier versión, esta versión ofrece algunas características propias de PostgreSQL que pueden ser útiles para el futuro.

La versión utilizada es un JDBC tipo4, esto quiere decir que se comunica con conexiones TCP/IP con la base de datos por tanto funcionará en cualquier plataforma cuyo protocolo del sistema acepte este tipo de conexiones, lo que hace que una vez compilada sea una plataforma independiente lista para ser usada sin complementos.

La API de JDBC se presenta como un conjunto de interfaces y métodos de gestión de manejadores de conexiones hacia un modelo especifico de bases de datos. Estos manejadores son los que implementan dichas interfaces para las distintas bases de datos soportadas por JDBC. En el siguiente apartado veremos las clases que hemos utilizado en nuestro proyecto y en apéndices explicaremos sus principales constructoras y funciones.

4.2 Clases utilizadas en nuestro proyecto

Connection

Esta clase representa una conexión física al servidor que contiene nuestra base de datos para que a través de ella se ejecuten las sentencias en el servidor. Para crear dicha conexión utiliza un objeto de la clase DriverManager que es el encargado de dada una URL a la base de datos a la que queremos conectarnos, encontrar el driver adecuado y usarlo para conectarnos a la base de datos.

-Formato de las URL:

El formato de las URL que acepta este objeto es el siguiente:

jdbc:[drivertype]:[database]

Donde [drivertype] representa el tipo de base de datos a la que queremos conectarnos, en nuestro caso postgresql.

La tercera parte, [database], representa la base de datos concreta a la que queremos conectarnos, hay varias formas de representar esta información:

databasename

//hostname/databasename

//hostname:portnumber/databasename

Si omitimos hostname o portnumber se utilizarán localhost por defecto y el puerto por defecto.

SQLException

Esta excepción es la que salta en caso de que encontremos algún problema al ejecutar sentencias en el servidor de la base de datos o al establecer la conexión.

Statement

Este objeto permite realizar consultas y modificaciones sobre nuestra base de datos, devolviéndonos el resultado de la sentencia ya sea indicándonos el número de filas modificadas o el resultado de una consulta. La sentencia que representa este objeto no es analizada para ver si es sintácticamente válida para el servidor que estamos usando sino que se pasa tal cual al servidor y este es el encargado de analizarla y ejecutarla si es posible o devolver el error correspondiente si no se puede.

PreparedStatement

Similar a Statement, se utiliza para representar sentencias SQL que son utilizadas múltiples veces con valores diferentes de manera que con el mismo objeto podemos ejecutar varias veces una sentencia con distintos valores. La ventaja principal de este objeto es que es precompilado, reduciendo la saturación provocada por parsear la sentencia SQL en cada ejecución.

ResultSet

Es una interfaz básica para el acceso a la información de nuestra base de datos. Con este objeto podemos acceder a las distintas filas devueltas por una consulta, así como acceder a los valores de las columnas. Los valores devueltos pueden ser de cualquier tipo que soporte PostgreSQL.

ResultSetMetaData

Este objeto representa la información detallada de una consulta. Es un objeto que se utiliza para encontrar información adicional a ResultSet, contiene información como el número de columnas, el nombre de estas, el tipo de las columnas, etc.

5. SAX - XERCES

5.1 Visión general

SAX (Simple API for Xml) es una API útil para lectura de ficheros xml y su mapeo en clases java. Es importante saber que SAX es una API, es decir, una manera de afrontar el problema y no una implementación. Nosotros hemos elegido la implementación de SAX que ofrece Apache, cuyo nombre es Xerces aunque existen otras muchas implementaciones y la diferencia entre ellas no es significativa a la hora de evaluar el rendimiento de nuestro proyecto.

A diferencia de otras API's (véase apartado SAX vs JDOM) SAX procesa el fichero xml según lo lee, sin crear estructuras adicionales en memoria más que la estrictamente necesaria para la lectura del fichero. SAX trabaja de acuerdo a eventos que se disparan al ocurrir ciertas cosas al procesar el fichero, y por tanto al realizar todas las acciones en una sola pasada mejoramos el rendimiento de nuestra aplicación.

En el siguiente apartado describiremos las principales clases de SAX utilizadas en el proyecto y su funcionamiento, y en el apéndice explicaremos el uso de sus funciones y constructoras, y en el caso de los métodos correspondientes a eventos sus parámetros si procede, puesto que el cuerpo es el que nosotros implementamos.

5.2 Clases utilizadas en nuestro proyecto

ContentHandler

Este interfaz es la más importante de SAX y es la encargada de llevar a cabo todo el procesamiento del archivo xml. Es el encargado de definir todos los eventos que pueden ocurrir al procesar el archivo xml y las acciones asociadas a dichos eventos. Por ello todas nuestras clases encargadas de procesar los archivos xml deberán implementar esta interfaz. En lugar de implementar esta interfaz, SAX ofrece ya una implementación "vacía" de manera que solo tenemos que extender dicha clase e implementar los métodos que nos interesen, esta clase es DefaultHandler pero no la utilizaremos en nuestro proyecto.

Una cualidad interesante de SAX es que podemos tener varias implementaciones de ContentHandler y pasar al parser uno u otro según la situación en tiempo de

ejecución para así modularizar el proceso y para poder tratar los nombres de etiquetas repetidos para entidades diferentes.
Los eventos que implementa ContentHandler son del estilo inicio/fin del documento, inicio/fin de un elemento, datos entre etiquetas, etc. Así cada evento tendrá un método que se ejecutará cada vez que dicho evento ocurra al leer el fichero xml (ver apéndices para ver más detalles de dichos métodos).

-Eventos:

Los eventos que existen son:

startDocument→Se produce al comenzar un archivo xml.

endDocument→Se produce al acabar un archivo xml.

startElement→Se produce cada vez que se empieza un nuevo elemento.

endElement→Se produce al acabar un elemento.

characters→Obtiene el valor comprendido entre una etiqueta de inicio y fin de elemento.

startPrefixMapping→ Inicio de prefijo

endPrefixMapping→Fin de la petición de inicio de prefijo

ignorableWhiteSpace→Se produce cuando se alcanza un numero de espacios en blanco determinado.

processingInstruction→Se produce cuando se encuentra con una instrucción de proceso xml, distinta a la declaración del documento xml.

skippedEntity→Este evento se produce cuando nos encontramos con una entidad para la cual no hay evento definido. Esto no ocurre en el parser de Xerces.

Attributes

Objeto que representa los atributos de una etiqueta concreta del archivo xml, haciéndolos accesible. Este objeto es creado automáticamente con sus valores en cada evento.

XMLReader

Interfaz de SAX que será la encargada de de leer los archivos xml, ésta es la interfaz cuya implementación cogeremos de Xerces. Es decir, esta interfaz es el parser de SAX.

SAXParser

Implementación del parser de Xerces que utilizaremos en nuestro proyecto. Es código libre de Apache Jakarta.

SAXException

Excepción que saltará con cualquier error que se produzca en SAX o en Xerces.

ErrorHandler

Interfaz de SAX encargada de tratar los eventos errores, warnings y errores fatales. Para utilizarla es necesario implementar dicha interfaz que solo tiene los 3 métodos correspondientes a estos 3 eventos especiales. Dichos eventos siempre vienen acompañados de una SAXException.

6. DESARROLLO/IMPLEMENTACION

6.1 Introducción

En este apartado y una vez estudiadas las tecnologías utilizadas en el proyecto, procederemos a explicar el desarrollo e implementación del proyecto. Para ello además de explicar lo que finalmente se ha desarrollado, explicaremos las decisiones tomadas frente a las desechadas.

El proyecto consta de seis aplicaciones, tres para la tecnología Lucene y tres para la tecnología PostgreSQL. Para cada tecnología las aplicaciones son, un indexador de los datos contenidos en los archivos xml, y dos motores de búsqueda para los datos indexados, uno con entorno gráfico y otro que solo funciona en modo texto para las máquina que no acepten entorno gráfico, como es el caso de la máquina marbore de universidad.

6.2 SAX vs JDOM

Un aspecto importante del indexador es qué tecnología utilizar para parsear los archivos xml que contienen los datos, para realizar tal tarea, las tecnologías más conocidas son SAX y JDOM, en un principio se eligió JDOM por su fácil manejo, llegándose incluso a haber hecho un indexador completo que utiliza esta tecnología combinada con Lucene, pero debido a problemas insalvables que esta tecnología presentaba aplicada a MEDLINE finalmente desechamos esta posibilidad y pasamos a SAX. A continuación expondremos cuales fueron dichos problemas.

El funcionamiento de JDOM es el siguiente, al parsear un archivo lee todo su contenido de una pasada y crea en memoria una estructura jerárquica en forma de árbol que representa la totalidad del archivo xml y que posteriormente puede usarse para acceder a la información para reutilizarla. El primer problema que presenta JDOM es que si se parsean archivos muy grandes, y los de MEDLINE lo son, dicha estructura ocupa muchísimo en memoria puesto que es una imagen de los datos más una gran cantidad de elementos de control, recorrido y manipulación. En concreto hemos comprobado que para los archivos más grandes de MEDLINE, que son de alrededor de 160 MB, la estructura ocupa en memoria más de 1 GB de memoria RAM. Otra desventaja de JDOM es que se deben hacer varios recorridos de todos

los datos, al menos una para crear la estructura y otra para acceder a los datos, con la consecuente perdida de tiempo.

Ahora veamos el funcionamiento de SAX y como para nuestros requerimientos supera ampliamente a JDOM. Como hemos visto anteriormente SAX reacciona a eventos mientras esta leyendo el archivo xml, esto quiere decir que no crea ninguna estructura en memoria salvo la estrictamente necesaria para leer y procesar el archivo, pero al no haber estructura con los datos, es necesario mapear en clases java los datos que nos interesen indexar configurando los eventos para tal efecto. Estas clases que representan toda la información que nos interesa sin embargo ocupa muchísimo menos espacio en memoria que la estructura creada por JDOM, en concreto, para los archivos de 160 MB de MEDLINE, mientras que con JDOM necesitábamos más de 1 GB de memoria con SAX necesitamos unos 150 MB de memoria RAM. En cuanto al tiempo aunque es cierto que con SAX se deben realizar dos recorridos como en JDOM, es más rápido un recorrido por nuestras clases java creadas con SAX que recorrer la totalidad del árbol con los datos del xml.

Por estas razones es por las que finalmente se desechó JDOM a favor SAX pese a que la complejidad de manejo de SAX es mucho mayor que la de JDOM.

6.3 Implementación

6.3.1 Indexadores

6.3.1.1 Clases comunes

En este apartado explicaremos la implementación de las clases comunes en ambos indexadores así como las decisiones tomadas para su implementación. En concreto, estas clases corresponden a la tecnología SAX y a las clases en las que se mapea la información, por que como es obvio la extracción de información es independiente de la tecnología que la utilizará.

MedlineCitation

Esta clase es la encargada de almacenar la información de cada MedlineCitation (referencia bibliográfica completa). Para tales efectos tiene tres atributos: PMID, un Objeto de la clase Article con la información referente al artículo al que hace referencia el MedlineCitation y un array de MeshHeading.

-Decisiones:

PMID pese a poder almacenarlo como un integer se decidió que fuera de tipo String para evitar conversiones de tipos innecesarias al indexarlo con Lucene y porque no tiene mucho sentido realizar operaciones matemáticas con un identificador único como es el pmid.

El atributo MeshHeadings en principio se pensó hacerlo tipo ArrayList puesto que el número de headings de una referencia bibliografica de MEDLINE es variable, pero debido a que el consumo de memoria de la aplicación se incrementó mucho al añadir los términos Mesh en el index (de menos de 50MB a unos 150MB) y con el fin de reducirlo todo lo posible, finalmente se decidió utilizar un array simple para guardar la información para posterior uso mientras que de manera temporal se utiliza un ArrayList hasta que se sabe cuantos headings tiene la MedlineCitation.

Article

Esta clase almacena la información relativa a un artículo concreto. Para ello tiene cuatro atributos para almacenar el titulo del artículo, su abstract, su año de publicación y el nombre de la revista que lo publicó.

-Decisiones:

La única decisión relevante para esta clase es el tipo del atributo que almacena el año de publicación, intentando que se realicen el mínimo número de conversiones de tipo sobre el valor pero conserve todas sus posibilidades de uso interesantes. Para ello, hay que considerar que las únicas operaciones que se realizarán sobre este valor son las de comparación. Lucene incorpora un tipo de querys para configurar rangos aun cuando el valor es de tipo String, con lo que desde el punto de vista de Lucene sería mejor que fuera tipo String. En cambio desde el punto de vista de PostgreSQL resultaría más sencillo hacerlo tipo int pero PostgreSQL también acepta comparación entre Strings considerando el orden alfabético, el cual para los años devuelve el mismo resultado que si fueran int. Es por ello que finalmente se mantuvo como tipo String.

MeshHeading

Esta clase almacena la información relativa a un único termino Mesh el cual esta compuesto por un descriptor y qualifiers. Para almacenar esta información utilizaremos un array de Topic. La estructura del array es la siguiente, el primer elemento se corresponde con el descriptor mientras que el resto de posiciones se corresponden con los qualifiers los cuales pueden ser cero o más.

-Decisiones:

Al igual que en la clase MedlineCitation se procede a guardar para posterior uso los topics en un array simple para ahorrar algo de espacio y se utiliza un ArrayList de manera temporal hasta que sabemos cuantos topics componen el termino Mesh.

Topic

Esta clase almacena la información perteneciente a un Topic de un termino Mesh, para ello tenemos un atributo para guardar su valor y otro que nos indica si es major o minor (es decir la importancia del termino).

Handler

Esta es la clase principal encargada de parsear el archivo xml y mapear la información en las clases anteriormente vistas. Decimos que es la clase principal porque realmente solo parsea el pmid y el OtherAbstract en caso de que lo haya, para parsear el resto de la información delega en otros handlers, por tanto su función es también la de gestionar que handler debe tomar el control en el proceso de parseo.

Esta clase aparte de implementar a ContentHandler (clase de SAX) también implementa ErrorHandler (clase de SAX) para mostrar con más detalle errores en caso de que se produzcan durante el proceso de parseo.

-Atributos importantes:

- instancias→ Es un ArrayList de MedlineCitations, en el almacenaremos todas las clases que produzcan el mapeo para su posterior utilización.
- parser→ Es el XMLReader que rige el proceso de parseo, es necesario para ceder el control a otros handlers.
- handlerArticle→ Handler encargado de parsear la información de los artículos
- headingHandler→Handler encargado de parsear la información relativa a la lista de términos Mesh.
- meshHeadingList→ ArrayList que servirá de almacenamiento temporal para la lista de términos Mesh.

-Eventos:

Ahora veamos como hemos configurado los eventos en este handler.

- startElement: En este handler solo nos interesan los comienzos de 3 elementos.

    - MedlineCitation: Al leer esta etiqueta creamos un nuevo objeto para almacenar una nueva MedlineCitation y lo añadimos a instancias.
    - Article: Al leer esta etiqueta creamos un nuevo objeto de la clase Article y se lo asignamos al objeto MedlineCitation actual para a continuación ceder el control del parseo al Handler encargado de parsear los artículos.
    - MeshHeadingList: Creamos un nuevo ArrayList que servirá como almacenamiento temporal de la lista de términos mesh para su posterior conversión en un array simple. También cede el control del parseo al Handler que parsea las listas de términos mesh.
- endElement: Nos interesan tres etiquetas.
    - PMID: recoge el valor del pmid que se ha obtenido con el evento characters.
    - AbstractText: A esta parte solo se accede cuando se está en OtherAbstract, el AbstractText normal lo parsea el handler de artículos. Si este abstract existe significa que el otro no existía y por tanto asignamos el valor recogido por characters al atributo abstract de objeto Article correspondiente al MedlineCitation actual.
    - MedlineCitation: Hemos acabado de parsear la referencia bibliográfica así que procedemos a transformar en un array simple la lista de términos Mesh y se la asignamos al MedlineCitation actual.
- characters: recoge el valor entre etiquetas y lo guarda temporalmente.

-Decisiones:

En lugar de dejar todo el trabajo de mapear y parsear el archivo xml a esta clase, se decidió hacer un handler distinto para cada entidad grande de las referencias bibliograficas para que el código no estuviera tan cargado y fuera más modular, así hicimos un Handler para los MedlineCitations, los Articles y para la lista de términos Mesh. Esta decisión concreta no responde a ninguna mejora en el rendimiento o capacidades de la aplicación, aunque el handler para artículos es imprescindible puesto que al haber muchas etiquetas que tienen por nombre Year necesitamos diferenciar de alguna manera cuando estamos en la fecha de

publicación y la única manera es delegar en otro Handler en un momento en el que ya no haya repeticiones de Year hasta que acabe ese Handler, y ese momento es al empezar la información de un articulo el cual solo contiene una fecha, la de publicación.

ArticleHandler

Esta clase es la encargada de parsear la información relativa a los artículos y como hemos visto es imprescindible para obtener la fecha de publicación. Es una de las clases en las que delega el proceso de parsear el Handler principal y por tanto debe ser capaz de devolver el control a dicho Handler. Implementa la Interfaz ContentHandler de SAX pero solo configuramos dos eventos de los posibles.

-Atributos importantes:

- handler→ Es de tipo Handler y contiene al Handler principal de manera que podamos devolverle el control dado el momento.
- parser→ De Tipo XMLReader, es necesario para devolver el control al handler principal.
- article→ Objeto de la clase Article, correspondiente al MedlineCitation actual y en el que se guardaran los datos que se obtengan con los eventos.

-Eventos:

- characters: recoge el valor entre etiquetas y lo guarda temporalmente.
- endElement: Nos interesan seis etiquetas:
    - ArticleTitle: Recoge el valor obtenido por characters y lo almacena en el objeto article.
    - AbstractText: Corresponde al abstract principal, no al OtherAbstract. Recoge el valor almacenado en characters y lo almacena en el objeto article.
    - Article: Hemos acabado de procesar la información del artículo así que devolvemos el control al Handler principal.
    - Year: Corresponde al año de publicación cuando esta en formato de 3 etiquetas para año, mes y día. Recogemos el valor del año y lo guardamos en el objeto article.

- Title: Corresponde al nombre de la revista que publicó el articulo, recogemos el valor y lo almacenamos en el objeto article.
- MedlineDate: corresponde al año de publicación cuando esta en formato de 1 etiqueta. Por ello extraemos solo el año del valor recogido por characters y lo almacenamos en el objeto article.

HeadingListHandler

Esta clase mapea en clases la información recogida en la lista de términos Mesh, esta clase podría estar integrada en la clase Handler si hubiésemos querido como hemos visto anteriormente. Es una de las clases en las que delega el proceso de parsear el Handler principal y por tanto debe ser capaz de devolver el control a dicho Handler. En esta clase configuramos 3 eventos de ContentHandler.

-Atributos importantes:

- parser→ De Tipo XMLReader, es necesario para devolver el control al Handler principal.
- handler→ Atributo que contiene el Handler principal al que se le debe devolver el control al acabar de mapear la información.
- headings→ Arraylist proveniente de la clase Handler y que sirve como contenedor temporal de los términos Mesh antes de su conversión a array simple.

-Eventos:

- characters: recoge el valor entre etiquetas y lo guarda temporalmente.
- startElement: Nos interesan 3 etiquetas.
  - MeshHeading: Creamos un nuevo ArrayList que contendrá la información de este termino Mesh.
  - DescriptorName: Esta etiqueta contiene un atributo que nos indica si el descriptor es major o minor. Accedemos a dicho atributo, si su valor Y entonces es major sino es minor. A continuación creamos un nuevo Topic y lo añadimos al ArrayList que representa el actual termino mesh, pero con valor vacío puesto que aun no lo hemos procesado, solo asignamos si es major o minor.

- QualifierName: Puede haber cero o más etiquetas de este tipo por cada termino mesh y siempre son posteriores al descriptor. Se procede de igual manera que con el descriptor.

- endElement: Nos interesan 4 etiquetas:
  - DescriptorName: Recogemos el valor de characters y lo guardamos en el Topic de la posición 0 del ArrayList que representa el termino Mesh.
  - QualifierName: Recogemos el valor que se obtuvo con characters y lo guardamos en el Topic que ocupa la ultima posición del ArrayList que representa el termino Mesh.
  - MeshHeading: Hemos acabado un termino Mesh así que transformamos el ArrayList que lo representa en un array simple y lo añadimos al ArrayList que representa la lista de términos Mesh.
  - MeshHeadingList: Fin de la lista de términos Mesh, por tanto devolvemos el control al Handler principal.

6.3.1.2 Indexador de Lucene

Main

Esta clase solamente es la encargada de recoger los argumentos de la aplicación, de crear el objeto de la clase indexer y ejecutar su método principal, run. Además de eso mide el tiempo de indexación.

-Decisiones:

Ahora hablaremos de los argumentos necesarios para ejecutar la aplicación. Queríamos hacer la aplicación lo más independiente y versátil posible, ya que como hemos visto las posibilidades de indexación de Lucene son varias, pero tampoco queríamos cargarla demasiado con argumentos, así que finalmente optamos por dejar constantes la mayoría de las características del proceso de indexación a nuestro criterio (se verán en el apartado siguiente). Los argumentos que finalmente recibe nuestra aplicación son: la ruta absoluta del directorio que contiene los archivos xml a indexar, la ruta absoluta del directorio donde se quiere crear el index y si se quiere ampliar el index o sobrescribir/crear uno.

indexer

Esta clase es la encargada de iniciar el proceso de obtención de los datos de los archivos xml y una vez obtenidos indexarlos. Debido a la importancia de esta clase explicaremos su secuencia de acciones. Pero primero veamos sus atributos.

-Atributos importantes:

- mLuceneIndexer→ Es de la clase IndexWriter de Lucene, y como hemos visto será la encargada de añadir documentos a nuestro index.
- reader→ Es de la clase XMLReader de SAX, y es la encargada de procesar los archivos xml de MEDLINE.
- instancias→ ArrayList que contiene los datos de un archivo xml.

-Secuencia de acciones:

- Abrimos el index, asignamos el parser de Xerces a SAX, creamos el handler principal y se lo asignamos al parser.
- Creamos una Lista con los nombres de todos los archivos que existen en el directorio de los datos.
- Para cada archivo si este tiene extensión ".xml", se procede a la extracción de los datos y se indexa.
- El proceso de indexación, extrae una a una las referencias contenidas en el atributo instancias, correspondientes únicamente a las referencias contenidas en el ultimo archivo parseado, crea un documento para cada referencia y lo añade al index. Si se consigue añadir al index se muestra por pantalla algunos datos representativos de la referencia indexada.
- Se muestra por pantalla el número de referencia bibliográfica indexada.
- Se optimiza el index y se cierra.

-Decisiones:

Primeramente decidimos que tipo de index íbamos a utilizar. Para ello teníamos la opción de multifile index y compound index. Como vimos en su momento, aunque el multifile index realiza el proceso de indexación más rápido que el compound index, no es adecuado para nuestra aplicación puesto que el multifile esta indicado para bases de datos pequeñas o medianas que requieren mucha actualización y sacrifican velocidad de búsqueda a cambio, y en principio nuestro index una vez

creado completamente tardará en ser actualizado. Por tanto nos interesa más el compound index que es más rápido con las búsquedas aunque tarde entre un 5% y un 10% más de tiempo en indexar. Además para bases de datos grandes el multifile index suele provocar un error de “too many files open”.

Posteriormente, había que decidir que Analyzer utilizar para insertar los datos en el index, elegimos el SimpleAnalyzer, ya que al indexar con este Analyzer conseguíamos que en el futuro nuestras búsquedas no fueran sensibles a mayúsculas y porque no nos interesaba eliminar las stop words ya que cualquier campo podía ser requerido para mostrarse en las búsquedas y un texto sin stop words carece de sentido.

Después había que decidir que valores dar al mergeFactor, maxMergeDocs y maxBufferedDocs, pero dichos parámetros son muy dependientes de la máquina en la que se ejecute la aplicación, es por ello que, tras comprobar que funcionaba bien en las máquinas a nuestra disposición, se dejaron los valores estándar que asigna Lucene.

También había que decidir en que momento optimizar el index puesto que cuando la base de datos alcanza gran tamaño este proceso puede durar varios minutos, la mejor opción es optimizar únicamente cuando sepamos que el index no se va a tocar por un tiempo, pero ello requeriría un argumento adicional para la aplicación que nos indicara si queremos o no optimizar, así que para no cargar de argumentos el comando de la aplicación se decidió que se optimizara el index cada vez que la aplicación acaba, esto supone cierta perdida de tiempo si vamos a añadir archivos nuevos al index cada muy poco tiempo, aunque como hemos mencionado esta perdida es solo de unos minutos.

Puesto que el consumo de memoria es relativamente elevado se decidió que una vez añadidos al index los datos de un archivo de MEDLINE se procediera a borrar el contenido del objeto instancias que contenía los datos del archivo, esto no es estrictamente una decisión puesto que si no se borraran estos datos, al parsear unos cuantos archivos MEDLINE nos quedaríamos sin memoria.

Por último, hablaremos de las decisiones tomadas en la forma de indexar cada dato, ya que como vimos Lucene nos ofrece múltiples posibilidades, para ello utilizamos el Objeto Field que nos permite personalizar la forma de tratar los datos a indexar. Puesto que todos los datos deberían de poder ser recuperados para mostrar su valor al realizar una búsqueda se decidió que todos los datos fueran almacenados en el

index (opción Field.Store.YES). El PMID está compuesto por números y como vimos el SimpleAnalyzer divide por tokens cada vez que encuentra un carácter que no corresponde a una letra con lo cual no tenía mucho sentido tokenizar el pmid puesto que el resultado sería el mismo que no tokenizarlo, además el pmid solo se compone de 8 caracteres numéricos que conforman una única entidad y es lógico que conformen un único token, con lo cual se decidió que no fuera analizado por el SimpleAnalyzer y por tanto que fuera almacenado tal cual (opción Field.Index.NO_NORMS). Para el resto de datos se decidió que los valores fueran tokenizados o analizados, puesto que es necesario para realizar búsquedas en ellos (opción Field.Index.TOKENIZED).

Para los términos Mesh, los cuales pueden ser un número variable, se decidió utilizar un field agregable, puesto que la definición de este es justamente lo que necesitábamos, un field multivaluado que permite búsquedas individuales para cada término almacenado en él. Así pues, se crearon dos fields para almacenar los valores que componen un término mesh, un field para los major y otro para los minor.

6.3.1.3 Indexador de PostgreSQL

Main

Al igual que su homologa del indexador de Lucene, esta clase solamente es la encargada de recoger los argumentos de la aplicación, de crear el objeto de la clase indexer, ejecutar su método principal run y de medir el tiempo de indexación.

-Decisiones:

Como vimos antes, queríamos hacer la aplicación lo más independiente y versátil posible, pero las posibilidades de indexación de PostgreSQL son mucho menos personalizables que con Lucene, así que se decidió poner los argumentos imprescindibles para hacer amena la utilización de la aplicación. Los argumentos que finalmente recibe nuestra aplicación son: la ruta absoluta del directorio que contiene los archivos xml a indexar, un usuario que pueda conectarse a la base de datos, su password, el nombre de la base de datos que se quiere crear/ampliar/sobrescribir y si se quiere sobrescribir/crear el index o ampliar.

indexer

Esta clase es muy similar a la del mismo nombre del indexador de Lucene, así es la encargada de iniciar el proceso de obtención de los datos de los archivos xml y una

vez obtenidos indexarlos en PostgreSQL. Debido a la importancia de esta clase explicaremos su secuencia de acciones. Pero primero veamos sus atributos.

-Atributos importantes:

- c→ Es de la clase Connection de JDBC, y como hemos visto será la representación de la conexión física con PostgreSQL y a través de la cual se ejecutarán las sentencias.
- reader→ Es de la clase XMLReader de SAX, y es la encargada de procesar los archivos xml de MEDLINE.
- instancias→ ArrayList que contiene los datos de un archivo xml.

-Secuencia de acciones:

- Primeramente debemos ver si se debe o no crear la base de datos.
- Después procederemos a conectarnos a la base de datos concreta especificada por parámetro y si acabamos de crear la base de datos crearemos las tablas necesarias para contener la información (veremos cuales son en el siguiente apartado).
- Asignamos el parser de Xerces a SAX, creamos el handler principal y se lo asignamos.
- Creamos una Lista con los nombres de todos los archivos que existen en el directorio de los datos.
- Para cada archivo si este tiene extensión ".xml", se procede a la extracción de los datos y se indexa.
- El proceso de indexación, extrae una a una las referencias contenidas en el atributo instancias, correspondientes únicamente a las referencias contenidas en el ultimo archivo parseado, se configuran las sentencias necesarias para ejecutar los datos en las tablas y se ejecutan dichas sentencias. Si se consigue añadir al index se muestra por pantalla algunos datos representativos de la referencia indexada.
- Se muestra por pantalla el número de referencias bibliográficas indexadas.
- Se cierra la conexión.

-Decisiones:

Primeramente debimos decidir como debíamos realizar el proceso de creación de la base de datos, y por consiguiente su proceso de reescritura en caso de que

exista. Para ello consideramos dos opciones y las dos fueron probadas y funcionaban correctamente, la primera consistía en que la base de datos sobre la que se iban a guardar los datos debía existir previamente a comenzar el proceso de indexación (por ejemplo, creándola con psql), y por tanto el indicar en el argumento correspondiente que queríamos crear el index desde cero equivalía a borrar las tablas que utilizamos en caso de que existieran y volverlas a crear. Este sistema funcionaba perfectamente pero analizándolo desde el punto de vista del uso, pensamos que era ciertamente incomodo tener que crear la base de datos aparte y que además si se ponía como nombre una base de datos que no tenia nada que ver con nuestra aplicación, esta creaba las tablas y añadía los datos a esa base de datos. Para evitar estas cosas se decidió que aunque era más complejo valía la pena hacer que fuera nuestra aplicación la que creara la base de datos. El problema para realizar esta opción radicaba en que para crear una base de datos necesitamos conectarnos a una base de datos, puesto que las conexiones de JDBC así lo exigen. Pero esta base de datos inicial o temporal debía de ser forzosamente una que existiera siempre en cualquier sistema PostgreSQL y el nombre de la base de datos inicial no es constante con lo cual esa no nos valía. Otro problema similar era que si debíamos borrar nuestra base de datos con los datos de MEDLINE, no podíamos estar conectada a ella. La solución a todos estos problemas fue utilizar como base de datos auxiliar la plantilla de PostgreSQL template1. Así si debemos crear o borrar-crear la base de datos que va a utilizar nuestra aplicación nos conectamos a template1 y realizamos la operación para luego desconectarnos y volvernos a conectar esta vez a la base de datos creada y entonces creamos las tablas para que estén contenidas en la base de datos correcta. Este es el sistema por el que finalmente hemos optado par el indexador de PostgreSQL.

Luego, hubo que decidir que enconding utilizar para la base de datos. Hemos elegido el encoding LATIN9 por estar especializado en el inglés, ya que MEDLINE está en inglés.

Al igual que en Lucene, como el consumo de memoria es elevado se decidió que una vez añadidos al index los datos de un archivo de MEDLINE se procediera a borrar el contenido del objeto instancias que contenía los datos del archivo para no incrementar la memoria requerida.

Por último veremos que tablas decidimos crear. Los términos Mesh pueden ser varios por cada referencia bibliográfica luego teníamos dos opciones para almacenar los términos mesh, la primera utilizar una estructura exclusiva de PostgreSQL, los array, pero resultó que estos enlentecían el proceso de indexación y búsqueda frente a la opción que ahora veremos y uno de los grandes problemas de PostgreSQL es el tiempo, aunque no el único, como veremos en Evaluación y rendimiento, con lo cual se desechó esta opción para intentar acelerar el proceso de indexación. La opción elegida fue utilizar una segunda tabla que relacionara los términos mesh con su referencia bibliográfica correspondiente.

En segundo lugar, había que decidir que restricciones se añadían a las tablas, teniendo en cuenta que cada una de ellas afectaría al tiempo de indexación. Así finalmente, solo incluimos las estrictamente necesarias, es decir las claves primarias, esenciales si se quiere realizar consultas en un tiempo razonable en tablas con millones de filas. También resultaba interesante obligar a que las filas insertadas en la tabla para los términos mesh tuvieran su valor de pmid existente en la tabla de referencias bibliográficas, puesto que no tiene mucho sentido incluir términos mesh de referencias bibliográficas sobra las que no se tiene constancia, pero finalmente esta comprobación se implementó en la clase indexer haciendo uso de las excepciones que ofrece JDBC, para así no cargar más las tablas. Finalmente las tablas quedaron de la siguiente manera.

- MedCitations (PMID text, Title text, Abstract text, PubYear text, JournalTitle text, PRIMARY KEY (PMID))

  La clave primaria es pmid ya que este es un identificador único de las referencias bibliográficas, luego es el candidato idóneo.

- MeshCitations (PMID text, mesh text ,Mayor text, PRIMARY KEY (PMID, mesh))

  Aquí la clave primaria es la dupla (pmid, mesh), ya que cada referencia bibliográfica puede tener varios mesh y por tanto el valor de pmid aparece repetido en la tabla al igual que los otros dos.

  También se tuvo en cuenta para crear las tablas una aplicación que se llama XMLPipeDB, que transforma un dtd en un esquema relacional sin embargo esta opción se desechó ya que a nosotros nos interesaban solo unos cuantos campos de los XML de medline no todos.

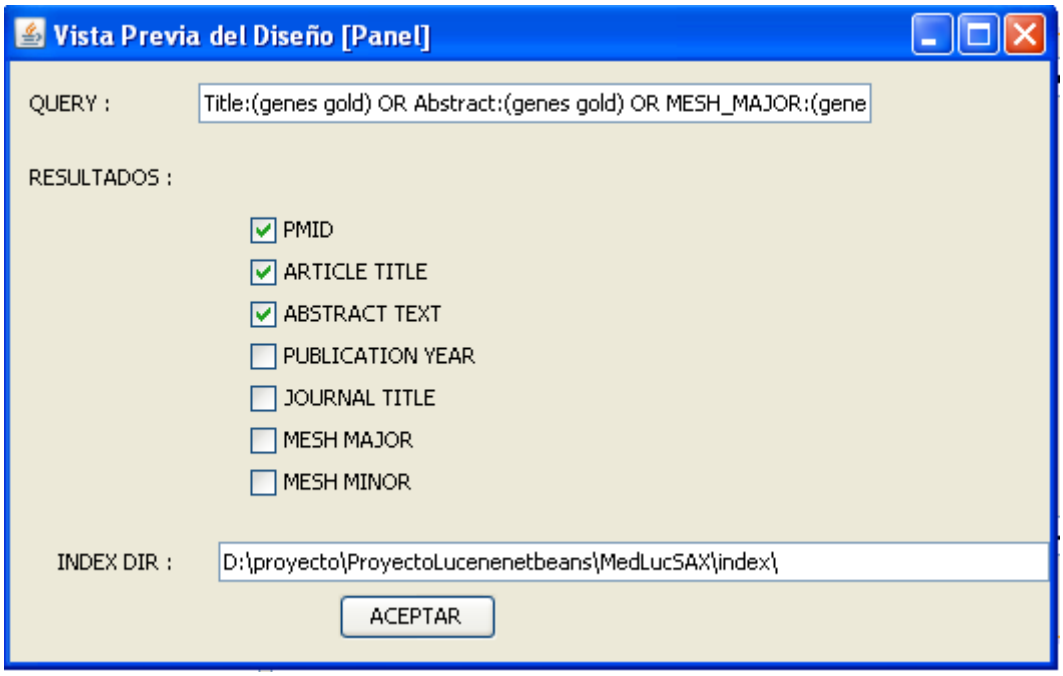
6.3.2 Motores de búsqueda

6.3.2.1 Motor de búsqueda de Lucene

Como hemos visto, hemos hecho dos motores de búsqueda para Lucene uno que recopila los datos de la consulta con entorno gráfico y otro que lo hace únicamente por texto, es decir, mediante los argumentos de la aplicación. Esto es debido a que hay máquinas, como marbore, que no aceptan ningún tipo de entorno gráfico. Así pues, la única clase que cambia entre los dos buscadores es la encargada de recopilar los datos, mientras que el motor de búsqueda propiamente dicho es idéntico en ambas aplicaciones. A continuación veremos las dos clases encargadas de recopilar los datos de la consulta y el motor de búsqueda.

Panel

Esta clase es la encargada de recopilar los datos de la consulta en el motor de búsqueda con entorno gráfico, y una vez obtenidos, crear el motor de búsqueda y llamar a su método principal para que realice la búsqueda. El aspecto del entorno gráfico es el siguiente:



Como vemos es bastante sencilla, solamente necesita que introduzcamos la query a realizar en sintaxis QueryParser de Lucene, los campos que queremos que sean mostrados por pantalla de los documentos que satisfagan dicha query y la ruta absoluta del index sobre el que debemos realizar la búsqueda.

-Decisiones:

El resultado de las búsquedas de Lucene siempre devuelven el Documento completo que satisface la query, pero normalmente al realizar una consulta solo nos interesa un número limitado de campos, es por lo que decidimos incluir un sistema para indicar explícitamente que campos de los documentos retornados queremos visualizar. Para ello decidimos que lo más sencillo que cumplía el objetivo era marcar los campos que queríamos que se devolvieran.

Main

Esta clase es la Homóloga a Panel pero para el motor de búsqueda textual, es decir, es la encargada de recopilar los datos de la consulta con sintaxis también de QueryParser, crear el motor de búsqueda y llamar a su método principal para que realice la búsqueda una vez obtenidos todos los datos.

-Decisiones:

Queríamos que la forma de indicar los datos de la consulta fuera similar a su homologa con entorno gráfico. Para ello, no supone ningún problema obtener la query y la ruta absoluta del index, basta con pasarlos por argumento a la aplicación. El principal problema estaba en como indicar que campos queríamos que se mostraran de los documentos resultado, este aspecto no se puede incluir en la query, ya que la sintaxis de las querys de Lucene no lo acepta. Las opciones eran mostrar todos los campos, mostrar siempre los mismos campos o inventarse algún sistema similar a lo que hicimos en el buscador con entorno gráfico. Como las dos primeras opciones no son nada buenas, decidimos inventar un sistema y el resultado fue indicar en los parámetros de la aplicación los campos que debían ser mostrados (ver comando en Apéndices) aunque esta solución cargaba bastante de parámetros el comando de ejecución y no es tan sencilla como con entorno gráfico.

MotorLucene

Esta clase es común a los dos buscadores y es la encargada de realizar la búsqueda en el index y mostrar por pantalla los resultados correspondientes. Además mide el tiempo que se tarda en abrir el index, procesar la query y realizar la búsqueda.

-Secuencia de acciones:

- Creamos un objeto IndexReader y abrimos el index con él y creamos un objeto IndexSearcher al que le asignamos el IndexReader que acabamos de crear.

- Creamos el objeto QueryParser y analizamos la query que hemos recibido por parámetro.
- Realizamos la búsqueda y una vez concluida calculamos el tiempo que hemos tardado hasta ahora.
- Mostramos por pantalla los resultados de los documentos requeridos por parámetro y mostramos también el tiempo calculado anteriormente.
- Cerramos el index.

-Decisiones:

La única decisión relevante era decidir si creábamos las Querys manualmente o utilizábamos el objeto de tipo QueryParser de Lucene para hacerlo. Finalmente decidimos usar QueryParser, puesto que además de funcionar bien, proporciona al usuario final una sintaxis que facilita la realización de consultas, de otra manera la interfaz para la introducción de querys se complicaría mucho obligando al usuario a conocer todos los objetos derivados de Query que ofrece Lucene de forma que elija que querys crear, con que parámetros y como combinarlas para crear la query final. Con QueryParser tenemos una sintaxis para construir expresiones que representan querys, QueryParser las analizará y convertirá automáticamente en el tipo de querys que corresponda y las combinará par formar la query final. Por tanto esta la opción que elegimos tras comprobar que su funcionamiento era correcto.

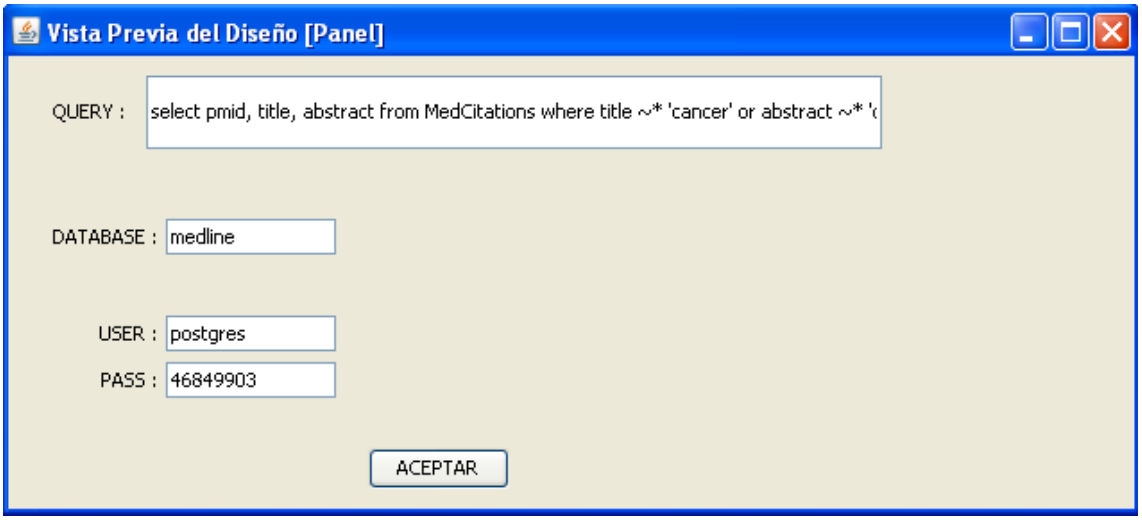
Una vez decidido que utilizaríamos QueryParser, había que decidir que Analyzer de Lucene utilizar para parsear la expresión que conforma la query, para ello hay que tener en cuenta que se utilizó el SimpleAnalyzer para crear el index y que por tanto todas las palabras en el index están en minúsculas. El Analyzer elegido es el StandardAnalyzer, ya que pone todo en minúsculas y además elimina las stop words las cuales no nos interesan a la hora de realizar una búsqueda.

6.3.2.2 Motor de búsqueda de PostgreSQL

Al igual que para Lucene, hemos hecho dos motores de búsqueda para PostgreSQL uno que recopila los datos de la consulta con entorno gráfico y otro que lo hace únicamente por texto, para que funcione en máquinas, como marbore, que no aceptan ningún tipo de entorno gráfico. Así pues, la única clase que cambia entre los dos buscadores es la que recopila los datos, mientras que el motor de búsqueda es el mismo en ambas aplicaciones. A continuación veremos las dos clases encargadas de recopilar los datos de la consulta y el motor de búsqueda.

Panel

Esta clase recopila los datos de la consulta en el motor de búsqueda con entorno gráfico, una vez obtenidos, crear el motor de búsqueda que es de tipo BusquedaSQL y llama a su método run() para que realice la búsqueda. El aspecto del entorno gráfico es el siguiente:



Como vemos solamente necesitamos indicar la query en sintaxis SQL, la base de datos sobre la que queremos buscar y un usuario un contraseña para acceder a dicha base de datos.

Main

Esta clase es la encargada de recoger los datos en el buscador textual de Postgresql. Crea un objeto de tipo BusquedaSQL y llama a su método run() para realizar la búsqueda. Los datos recogidos son pasados como parámetros de la aplicación y son los mismos que en el buscador con entorno gráfico.

BusquedaSQL

Esta clase constituye el motor de búsqueda para PostgreSQL y es una clase común en ambos buscadores. Es la encargada de mandar la búsqueda al servidor de Postgresql y mostrar los resultados de ésta. También mide el tiempo que tarda en crear la conexión y realizar la búsqueda.

-Secuencia de acciones:

- Crea un objeto de tipo Connection que representa la conexión física al servidor de PostgreSQL.
- Ejecuta la query y una vez concluida mide el tiempo que transcurrido hasta el momento.

- Muestra por pantalla las filas que componen el resultado de la query, así como el tiempo que se calculó previamente.
- Cierra la conexión con el servidor.

7. EVALUACION Y RENDIMIENTO

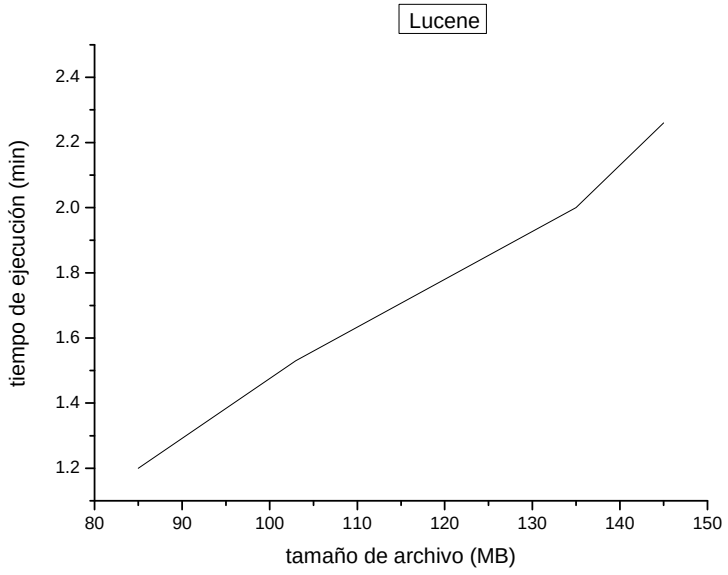
En este apartado analizaremos el rendimiento de cada tecnología y evaluaremos las posibilidades que ofrece cada una, para finalmente decidir qué tecnología es superior para el tratamiento de MEDLINE.

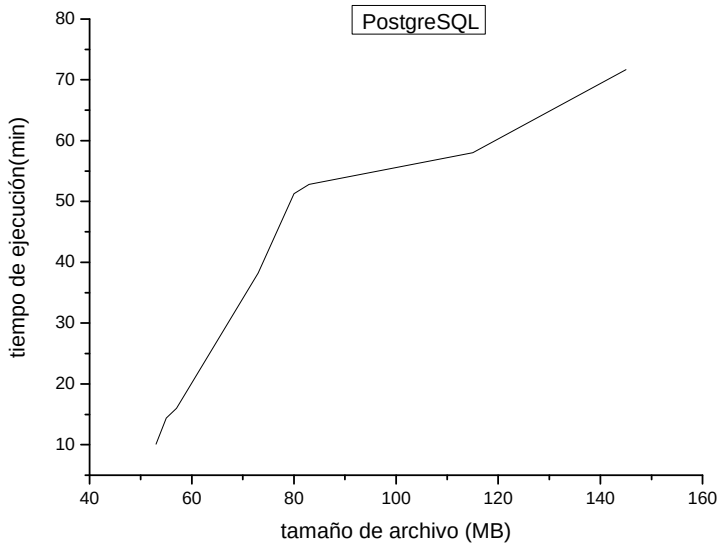
7.1 Rendimiento

A continuación analizaremos los tiempos de ejecución de los indexadores de ambas tecnologías frente al tamaño de los archivos xml de MEDLINE. También analizaremos los tiempos de búsqueda y los problemas de memoria que se tuvieron con los indexadores.

Tiempos de ejecución de los indexadores

Primero veamos unas gráficas ilustrativas de los tiempos necesarios para indexar:





A simple vista podemos observar varias cosas. La primera es que en todo momento el tiempo necesario para indexar un archivo de un tamaño concreto es siempre muy superior en PostgreSQL que en Lucene, tanto es así que los archivos más pequeños tardan en ser procesados por el indexador de PostgreSQL unas 4.46 veces más que el archivo más grande por Lucene. También observamos que mientras que la gráfica de Lucene experimenta un crecimiento más o menos constante del tiempo de indexación, la gráfica de PostgreSQL es mucho más irregular siendo al principio el crecimiento del tiempo de ejecución muy pronunciado para luego reducir su pendiente. De esta forma la razón entre el tiempo que tarda cada tecnología para el archivo más pequeño y el más grande es:

-Lucene→ 1.88

-PostgreSQL→ 7.1

Estos valores denotan un crecimiento medio mucho mayor del tiempo ejecución relativo de cada tecnología.

Así mientras que con Lucene se tardó exactamente 14 horas 31 minutos y 19 segundos en indexar todos los datos de MEDLINE, con PostgreSQL estimamos que se tardaría aproximadamente 487 horas 23 minutos. En otras palabras el indexador de PostgreSQL tardaría 33.6 veces más en ejecutarse para todos los datos de MEDLINE. Así que como vemos Lucene es muy superior en este aspecto.

Tiempos de ejecución de los buscadores

Ahora veamos cuales son los tiempos necesarios para realizar búsquedas en cada tecnología. Debido a que como hemos visto el tiempo de indexación con

PostgreSQL es inmenso (aproximadamente 20 días) para testar los tiempos de búsqueda se creó un index auxiliar de Lucene y una base de datos auxiliar de PostgreSQL con aproximadamente el 30% de los datos que conforman la totalidad de MEDLINE. Veamos un ejemplo para comparar las velocidades de búsqueda:
La query consistirá en buscar “hair” en el titulo del artículo, su abstract, y/o sus términos mesh.

- Query Lucene→ Title:hair OR Abstract:hair OR MESH_MAJOR:hair OR MESH_MINOR:hair
- Query PostgreSQL→select pmid, title, abstract from MedCitations where title ~* 'hair' or abstract ~* 'hair' or exists(select * from MeshCitations where MedCitations.pmid=MeshCitations.pmid and mesh ~* 'hair')

El motor de búsqueda de Lucene retornó 7546 documentos y tardó en realizar la consulta sobre el index 22 milisegundos y el motor de búsqueda de PostgreSQL retornó para la misma consulta 11892 filas en 454 segundos 7 milisegundos.
La diferencia de número de resultados es que no existe equivalente exacto entre los operadores de ambas tecnologías, así mientras que Lucene solo acepta la ocurrencia de la palabra “hair”, el operador de PostgreSQL acepta toda cadena de caracteres que contenga “hair”.
A primera vista se observa que la diferencia de rendimiento es abrumadora, pero calculemos cuanto mejor es Lucene:

(454*1000+7)/22= 20636.68

Como vemos Lucene es 20 mil veces más rápido, así pues no hay lugar a discusiones, Lucene es mejor.
Aun así hagamos una estimación de cuanto tardaría para toda la base de datos completa de MEDLINE. Se ha comprobado que la misma query para la totalidad del index tarda 2 segundos y 22 milisegundos con Lucene así pues una estimación de acuerdo a lo calculado anteriormente para PostgreSQL sería 41727 segundos 37 milisegundos. Este rendimiento tan malo de las querys de PostgreSQL es debido principalmente a que se utilizan las dos tablas y a que las operaciones con expresiones regulares son bastante lentas, si ponemos una query más sencilla (por ejemplo que solo implique la tabla de MedCitations y 4 operaciones con expresiones regulares) el tiempo se reduce a aproximadamente 40 segundos para el 30% de los datos y 2943 segundos para el total, aunque depende del numero de filas seleccionadas.

Problemas con la memoria

Un problema que se encontró al mapear en clases la información era que al contener los archivos de XML de MEDLINE 30000 referencias cada uno era necesario crear muchísimos objetos para mapear la información en clases java, aún cuando se borraban dichos objetos cada vez que se procesaba un nuevo archivo. En concreto, es necesario crear del orden de 700000 objetos aunque este numero depende mucho del número de términos Mesh que contienen las referencias del archivo procesado (hemos considerado una media de 10 términos mesh con 1 qualifier cada uno para estimar este número), el número de objetos creados sin tener en cuenta los términos mesh es de 60.000 exactos, como vemos la inclusión de términos mesh y el hecho de que el número de ellos sea variable tanto en los términos en sí como en los qualifiers disparó el número de objeto y por tanto la demanda de memoria. La solución fue simplemente asignar más memoria a la máquina java con las opciones de ejecución -Xmx y -Xms.

Espacio en disco

En cuanto al espacio que ocupan en el disco duro las bases de datos no ha habido ningún problema, siendo el espacio requerido con la tecnología Lucene de 18.734 MB aproximadamente. Mientras que para PostgreSQL no se ha podido medir dicho valor. El espacio de las fuentes de datos de MEDLINE (archivos .XML) es 56.775 MB

7.2 Evaluación

A continuación procederemos a evaluar las capacidades de cada tecnología para indexar y para realizar búsquedas.

Capacidades de indexación

Como hemos visto las capacidades de indexación generales son superiores en PostgreSQL, pero ello solo si tenemos en cuenta todos los tipos que ofrece y las posibilidades para cada tipo que ofrece, pero puesto que la información de MEDLINE es únicamente textual compararemos las capacidades de ambas tecnologías únicamente en esta rama. De esta forma ya no está tan claro cual ofrece más capacidades. Veamos que ofrece y que le falta a cada una.

Lucene ofrece cualidades interesantes para la indexación, la más significativa es la posibilidad de personalizar el uso de memoria física y de disco duro para nuestro index con las variables MergeFactor, MaxMergesDocs, MinMergesDocs y

eligiendo el tipo de estructura física del index (multifiles y compound index) lo que nos permite, optimizar al máximo el rendimiento una vez sepamos en que máquina vamos a ejecutar el proceso, esta cualidad no está presente en PostgreSQL. El uso de analizadores para procesar el texto también nos permite personalizar la forma en que los valores textuales son procesados de manera bastante completa que postgreSQL la cual procesa los valores textuales de una única manera. La forma de guardar la información de Lucene en forma de documentos que poseen campos o fields y valores para ellos es mucho más similar y adecuada a la estructura XML de los archivos de MEDLINE que las tablas de PostgreSQL, así por ejemplo podemos guardar todos los términos mesh de una referencia en el mismo documento que el resto de la información de la referencia mientras que como hemos visto con PostgreSQL hemos tenido que utilizar una tabla extra para almacenar estos valores con la consiguiente repetición de valores de los pmid en muchas filas de esta tabla auxiliar para enlazar todos los términos mesh con la referencia que le corresponde.
Pero PostgreSQL también ofrece posibilidades que Lucene no tiene, una de ellas es que permite añadir restricciones automáticas a los valores que son procesados, la más interesante es que comprueba automáticamente que no haya referencias repetidas gracias a las claves primarias, pero estas restricciones pueden hacer más lento el proceso de indexación. En Lucene aunque las restricciones no existen, al ser Lucene una librería java que por tanto debe ir incorporada a una aplicación, podemos hacer cualquier restricción que sea necesaria manualmente.
Otro aspecto relacionado con los datos es que PostgreSQL permite actualizar los datos de sus tablas mientras que Lucene no ofrece esta posibilidad y para actualizar es necesario borrar los datos y reinsertarlos actualizados. PostgreSQL también ofrece la posibilidad de restringir el acceso automáticamente a la base de datos mientras que Lucene no posee esta cualidad.
Por último PostgreSQL, optimiza sus tablas cada vez que se inserta un valor en ellas con la consecuente perdida de tiempo cuando se insertan grandes cantidades de filas como es nuestro caso, mientras que Lucene permite elegir el momento en que será optimizada la estructura que conforma el index, ahorrando mucho tiempo si solo optimizamos una vez añadidos los datos.

Capacidades de búsqueda

A continuación veremos las posibilidades de búsqueda que ofrece cada tecnología, pero al igual que anteriormente solo tendremos en cuenta aquellas que afecten el tratamiento de valores textuales que son los que nos interesan.

PostgreSQL ofrece muchísimas funciones y operadores para transformar/operar los valores textuales mientras que Lucene no ofrece ninguna, sin embargo como hemos visto podemos implementarlas aparte si es necesario. PostgreSQL también ofrece la posibilidad de crear índices para intentar mejorar el rendimiento de sus búsquedas, mientras que Lucene no ofrece tampoco esta posibilidad.

La sintaxis para querys que ofrece PostgreSQL es ciertamente más completa que la de Lucene así nos permite por ejemplo, además de realizar la query, indicar como se deben ordenar los resultados en el misma expresión que conforma la query, mientras que la sintaxis de Lucene no permite esto y para ordenar los resultados es necesario configurar un objeto de tipo sort aparte de la query. Lucene además si queremos restringir la búsqueda a un subconjunto de valores, es decir, a los resultados de otra query, es necesario configurar un objeto de tipo Filter aparte, mientras que la sintaxis SQL permite incluir esta información mediante el uso de subquerys en la expresión que forma la query. Como vemos, en este aspecto aunque PostgreSQL ofrece quizá más facilidades, Lucene no se queda atrás ni mucho menos. Lucene además con el uso de analizadores permite tratar el valor textual de la query automáticamente para realizar la búsqueda, así por ejemplo borra las stop words si lo estimamos necesario y PostgreSQL lo único que hace es poner en minúsculas el texto.

Pero donde Lucene supera ampliamente a PostgreSQL es en el tipo de query que permite realizar, ello es debido a que como Lucene solamente trabaja con valores textuales todas sus querys están orientadas a trabajar con texto mientras que con PostgreSQL solo podemos realizar una pequeña parcela de las querys de Lucene utilizando las expresiones regulares para encajar con texto. Pero Querys como las PhraseQuerys, FuzzyQuerys o SpanQuery son imposibles de realizar con PostgreSQL y resultan muy interesantes, pero además la mayoría del resto de Querys que ofrece Lucene son muy complicadas de realizar con las expresiones regulares de postgreSQL resultando prácticamente imposible realizarlas. En otras palabras, las expresiones regulares de PostgreSQL solo son adecuadas para querys muy sencillas.

8. CONCLUSIONES Y TRABAJOS FUTUROS

8.1 Conclusión

Tras el análisis de las tecnologías parece bastante claro que Lucene es mucho más apropiada para manejar la base de datos de MEDLINE que PostgreSQL, ya que está especializada en bases de datos textuales y por tanto ofrece mejores posibilidades para tratar los datos y para realizar búsquedas y porque además supera muy ampliamente en rendimiento a PostgreSQL.

Así pues hemos procedido a crear el index de Lucene con todos los archivos que componen MEDLINE extrayendo de cada referencia bibliográfica el pmid, titulo del articulo al que se hace referencia, año de publicación, nombre de la revista que lo publicó, abstract y los términos mesh clasificados en importantes (major) y poco importantes (minor).

Requerimientos

Al estar implementadas en lenguaje java las aplicaciones pueden funcionar en cualquier sistema operativo mientras se tenga la máquina java correspondiente. La máquina java necesaria para ejecutar la aplicación es la JRE 1.6 o superior, y para la aplicación con unos 200 MB de memoria RAM debería ser más que suficiente. Si nuestra máquina no esta configurada para utilizar jre 1.6 basta ejecutar los siguientes comandos:

export JAVA_HOME=directorio del jre1.6

export PATH=$JAVA_HOME/bin:$PATH

8.2 Trabajos futuros

Una vez hemos visto que Lucene es superior y la hemos elegido como la tecnología que se utilizará con la base de datos de MEDLINE resultaría interesante añadir ciertas capacidades que no han sido incluidas en este proyecto por falta de tiempo y que no eran estrictamente necesarias para la evaluación de las tecnologías y también resultaría interesante una vez conocida la máquina definitiva en la que se ejecutarían las aplicaciones cambiar algunas cosas.

Indexador

- Una vez conocida la máquina en la que se ejecutará el indexador de Lucene resultaría interesante evaluar los valores que optimicen al máximo nuestro proceso con mergeFactor, maxMergeDocs y maxBufferedDocs.

Motor de búsqueda

- Aunque las posibilidades que ofrece la sintaxis de QueryParser, que es la que utilizamos en nuestra aplicación, es bastante completa, ésta se puede ampliar para que utilice filtros (Filter) y ordenamiento de los resultados (Sort). Tanto los filtros como los Sort deben ser configurados aparte como complementos de la query principal.

9. BIBLIOGRAFIA

9.1 Bibliografía utilizada

- Eric Hatcher y Otis Gospodnetic, "Lucene in Action", Manning Publications co., 2004, ISBN: 1932394281
- John C. Worsley y Joshua D. Drake, “Practical PostgreSQL”, O’reilly, 2002.
- Tutorial para MEDLINE de Lingpipe, http://www.alias-i.com/lingpipe/demos/tutorial/medline/read-me.html
- DTD de MEDLINE, http://www.nlm.nih.gov/bsd/licensee/data_elements_doc.html
- Explicación detallada de los campos que componen los XML de MEDLINE, http://www.nlm.nih.gov/bsd/licensee/elements_descriptions.html#status_value
- Opciones y experiencias al indexer MEDLINE, http://www.nodalpoint.org/2006/06/07/medline_xml_to_database_parser
- XMLpipeDB, http://xmlpipedb.cs.lmu.edu/index.shtml
- Manual de usuario de PostgreSQL, http://es.tldp.org/Postgresql-es/web/navegable/user/user.html
- Distintos manuales para PostgreSQL, http://www.lawebdelprogramador.com/cursos/mostrar.php?id=72&texto=PostgreSQL
- Manual de introducción a XML, http://recursos.dotnetclubs.com/sevilla/Aportaciones/IntroduccionAXML.pdf
- Mapeo de XML a Java 1, http://www.javahispano.org/articles.article.action?id=2
- Mapeo de XML a Java 2, http://www.javahispano.org/articles.article.action?id=11
- “Tools for loading MEDLINE into a local relational database”, Diane E. Oliver, Gaurav Bhalotia, Ariel S. Schwartz, Russ B. Altman, Marti A. Hearst.

9.2 Palabras clave

MEDLINE, Lucene, PostgreSQL, Parser, Mapear, XML, base de datos.

10. APENDICES

10.1 Funciones y constructoras

10.1.1 Lucene

IndexWriter

-IndexWriter (String, Analyzer, Boolean)

-IndexWriter (Directory, Analyzer, Boolean)

El primer argumento corresponde la URL donde se creara el index.

El segundo argumento es el Analyzer con el que queremos que se cree el index.

El tercer argumento indica si queremos que se sobrescriba/crea el index que esta en la URL indicada (true) o si queremos añadir mas documentos a un index ya existente ubicado en la URL dada (false).

infoStream

Variable publica de la clase IndexWriter que nos permite indicar donde queremos que nos muestre la información del proceso de indexación. Por ejemplo System.Out.

setUseCompoundFile

-setUseCompoundFile(bolean)

Método de la clase IndexWriter que nos permite indicar si queremos que el index que vamos a crear sea multiFile (false) o compuesto (true).

optimize

Método de la clase IndexWriter que optimiza el index uniendo lo más posible los segmentos creados. No recibe argumentos.

close

Método de la clase IndexWriter, IndexSearcher y IndexReader que cierra dicho Objeto liberando a su vez el cerrojo para que el index vuelva a ser accesible. No recibe argumentos. En el caso de IndexReader al cerrar el objeto se borran aquellos documentos que estaban marcados como borrables.

addDocument

-addDocument (Document)

Método de la clase IndexWriter que añade un documento dado por parámetro al index.

add

add(Field)

Método de la Document que añade un Field a un documento.

get

-get(String)

Método de la clase Document que retorna el valor del Field cuyo nombre es el pasado por parámetro de ese documento.

setBoost

-setBoost(int)

Método de la clase Document y de la clase Field que fija el valor del boost de documento o de un Field concreto.

Field

-Field (String, String)

-Field (String, String, org.apache.lucene.document.field.store, org.apache.lucene.document.field.index)

Constructora de la clase Field que corresponde a un campo y valor respectivamente de un documento. Puede recibir dos argumentos extra que indican si el field debe ser guardado y/o indexado y de que manera.

Aunque las Subclases de Field (Keyword, UnStored, etc) ya no se utilizan en las versiones mas avanzadas de Lucene todas sus constructoras reciben dos argumentos que son la tupla campo-valor.

IndexSearcher

-IndexSearcher(String)

-IndexSearcher(Directory)

Constructora de la clase IndexSearcher que recibe por parámetro la ubicación del index que queremos abrir.

-IndexSearcher(IndexReader)

Recibe por parámetro un objeto de la clase IndexReader que es el encargado de abrir el index.

search

-search(Query)

Método de la clase IndexSearcher, que realiza la búsqueda de una query concreta en el index y retorna un objeto de la clase Hits con los resultados de la búsqueda.

-search (Query, Filter)

Realiza la búsqueda de una query concreta en el subrango correspondiente al Filter y retorna el objeto de la clase Hits con los resultados de la búsqueda.

-search (Query, Sort)

Hace la búsqueda de la query ordenando los resultados de acuerdo a lo indicado en el objeto Sort, y retorna el objeto de tipo Hits con los resultados.

Sort

-Sort()

Crea el Sort por defecto

-Sort(String)

Crea el sort de acuerdo a un campo que recibe por parámetro

-Sort(String, boolean)

Crea el sort de acuerdo a un campo que recibe por parámetro, y el booleano indica si se debe o no invertir el orden natural.

-Sort(SortField [])

Crea el sort de acuerdo a un conjunto de campos.

SortField

-SortField(String)

Crea un sortField recibiendo el nombre de un campo que se utilizará para la ordenación

-SortField(String, sortField.type, boolean)

Igual que el anterior solo que se permite indicar el tipo de lo que contiene el campo, y si se debe invertir el orden de ordenamiento cuando se utilice este campo para ordenar.
Esta clase tiene algunos valores predeterminados para ordenar como FIELD_SCORE

DateFilter

-DateFilter(String , Date, Date)
Crea un filtro utilizando como rango las dos Date en el Field String.

before

-before(String, Date)
Método estático de la clase DateFilter, que devuelve un DateFilter que utiliza el siguiente intervalo (-infinito, Date)

after

-before(String, Date)
Método estático de la clase DateFilter, que devuelve un DateFilter que utiliza el siguiente intervalo (Date, infinito)

QueryFilter

-QueryFilter(Query)
Constructora que construye un Filtro para que restrinja el rango de documentos a los resultados de la query pasada por parámetro.

CachingWrapperFilter

-CachingWrapperFilter(Filter)
Constructora que recibe por parámetro el Filtro al que hace referencia.

QueryParser

-QueryParser (String, Analyzer)
Constructora de QueryParser que recibe por parámetro el field por defecto que se utilizará en las búsquedas y el Analyzer que se usara para analizar las expresiones de las querys.

parse

parse (String)

Método de la clase QueryParser que transforma una expresión que representa una query en un Objeto de tipo Query y lo retorna.

setOperator

-setOperator(QueryParser.variable)

Método de la clase QueryParser que configura el operador por defecto para las expresiones recibidas. Por defecto este operador es OR.

length

Método de la clase Hits que devuelve el número de documentos que satisfacían la query que creo el objeto Hits.

doc

-doc(int)

Método de la clase Hits que retorna el documento que ocupa la posición que indica su parámetro. Siendo el primero el 0.

id

-id(int)

Método de la clase Hits que devuelve el número interno del documento que ocupa la posición del entero pasado por parámetro en Hits.

score

-score(int)

Método de la clase Hits que devuelve el score normalizado del documento que ocupa la posición pasada por parámetro. El score se normaliza de acuerdo al valor más alto de todos los documentos contenidos en Hits.

Term

-Term(String, String)

Constructora de la clase Term que recibe una tupla campo-valor como parámetro.

TermQuery

-TermQuery (Term)

Constructora de TermQuery que es subclase de Query, que recibe por parámetro el término que constituye la query.

RangeQuery

-RangeQuery(Term, Term, boolean)

Constructora que recibe por parámetro los dos términos que definen el intervalo y un booleano que indica si los términos están incluidos (true) o excluidos (false) del intervalo.

PrefixQuery

-PrefixQuery(Term)

Constructora que recibe por parámetro el término que define la query.

BooleanQuery.add

-add(Query, boolean, boolean)

Método de la clase BooleanQuery que añade una cláusula a la query. Los dos parámetros booleanos indican como se debe añadir la cláusula, así e. primer parámetro corresponde a required y el segundo a prohibited de manera que son posibles las siguientes combinaciones:

false-false→ cláusula opcional

false-true→ cláusula prohibida

true-false→ cláusula obligatoria

true-true→ inválido.

PhraseQuery

-PhraseQuery (Term)

Constructora que recibe el Term que conforma la Query

setSlop

-setSlop(int)

Método de la clase PhraseQuery y PhrasePrefixQuery que asigna el flop a la Query.

WildCardQuery

-WildQuery(Term)

Constructora que recibe por parámetro el término que conforma la query.

FuzzyQuery

-FuzzyQuery(Term)

Constructora que recibe por parámetro el término que conforma la query.

PhrasePrefixQuery.add

-add(Term[])

-add(Term)

Añade como palabras que pueden encajar en una posición concreta determinada por el orden de llamadas add, el/los termino/s pasado/s por parámetro.

open

-open(String)

-open(Directory)

Método Estático de la clase IndexReader que abre el index ubicado en el argumento que recibe y que devuelve un objeto de la clase IndexReader.

maxDocs

Método de la clase IndexReader que devuelve el número interno del próximo documento disponible. No recibe parámetros.

numDocs

Método de la clase IndexReader que devuelve el número de documentos que contiene un index. No recibe parámetros.

delete

-delete(int)

Método de la clase IndexReader que marca como borrable el documento del index cuyo numero interno es el recibido por parámetro.

-delete(term)

Marca como borrables todos los documentos que contengan el término pasado por parámetro.

isDeleted

-isDeleted(int)

Método de la clase IndexReader que nos indica el estado en el que se encuentra el documento cuyo numero interno es el que recibe por parámetro devolviendo true si esta marcado como borrable o false en caso contrario.

hasDeletions

Método de la clase IndexReader que nos devuelve true si existe algún documento en el index marcado como borrable y false en caso contrario. No recibe argumentos.

10.1.2 JDBC

getConnection

-getConnection(String, String, String)

Método estático de la clase DriverManager que crea una conexión a una base de datos y devuelve un Objeto Connection que representa dicha conexión.
El primer argumento es la URL que representa la base de datos a la que nos queremos conectar, el segundo es el usuario y el tercero la contraseña necesarios para acceder al servidor que contiene la base de datos.

close

Método de la clase Connection que cierra de forma segura la conexión al servidor. No recibe argumentos.

prepareStatement

-prepareStatement(String)

Método de la clase Connection que crea un objeto de la clase PreparedStatement que representa una sentencia SQL sin valores concretos, representados por ‘?’, correspondiente al argumento que recibe y lo devuelve.

setString

-setString(int, String)

Método de la clase PreparedStatement que asigna el valor pasado por parámetro al valor cuya posición es el int en la PreparedStatement.

Existen métodos similares para cada tipo permitido como setInt, etc.

PreparedStatement.executeUpdate

Método de la clase PreparedStatement que manda al servidor de la base de datos la sentencia SQL que representa, dicha sentencia puede ser cualquiera menos una consulta y devuelve el número de filas afectadas por la sentencia.

Statement.executeUpdate

-executeUpdate(String)

Método de la clase Statement que manda al servidor de la base de datos la sentencia SQL que representa su parámetro, dicha sentencia puede ser cualquiera menos una consulta y devuelve el número de filas afectadas por la sentencia.

createStatement

Método de la clase Connection que crea un objeto de la clase Statement que representa una sentencia SQL y lo devuelve.

Statement.executeQuery

-executeQuery(String)

Método de la clase Statement que manda al servidor la sentencia SQL representada por su parámetro, que debe ser necesariamente una consulta y devuelve un objeto de la clase ResultSet con los resultados de dicha query.

next

Método de la clase ResultSet que coloca su cursor interno en la siguiente fila si la hay y devuelve true si tiene éxito y false si no quedan filas por seleccionar. Inicialmente dicho cursor se encuentra en la posición anterior a la primera fila.

getString

-getString(int)

Método de la clase ResultSet que devuelve el valor contenido en la columna cuya posición es indicada por el parámetro, de la fila que es apuntada por el cursor. Existen métodos similares para cada tipo aceptado por PostgreSQL como getInt, etc.

getMetaData

Método de la clase ResultSet que devuelve su objeto de la clase ResultSetMetaData asociado.

getColumnCount

Método de la clase ResultSetMetaData que devuelve el número de columnas que existen en las filas que contiene su ResultSet asociado.

getColumnName

-getColumnName(int)

Método de la clase ResultSetMetaData que devuelve el nombre de la columna cuya posición es int.

10.1.3 SAX-XERCES

SAXParser

Constructora del parser de Xerces para SAX. Corresponde al objeto XMLReader de SAX y por tanto este debe ser creado con esta constructora:

```
XMLReader parser= new SAXParser();
```

setContentHandler

-setContentHandler(ContentHandler)

Método de la clase XMLReader que nos permite asignar el Handler para tratar los eventos con él que se pasa por parámetro.

setErrorHandler

-setErrorHandler(ErrorHandler)

Método de la clase XMLReader que nos permite asignar el Handler para tratar los errores y warnings con él que se pasa por parámetro.

parse

-parse(String)

Método de la clase XMLReader que realiza el proceso del archivo xml cuya ruta absoluta es la pasada por parámetro.

startElement

-startElement(String, String, String, Attributes)

Método de la clase ContentHandler. Los argumentos son, por orden, la dirección URI del espacio de nombres asociado al elemento, el nombre del elemento sin el prefijo del espacio de nombres, el nombre del elemento en la versión 1.0 de la especificación del xml y los atributos que contiene la etiqueta.

endElement

-endElement(String, String, String)

Método de la clase ContentHandler. Los argumentos son, por orden, la dirección URI del espacio de nombres asociado al elemento, el nombre del elemento sin el prefijo del espacio de nombres y el nombre del elemento en la versión 1.0 de la especificación del xml.

characters

-characters(char[], int, int)

Método de la clase ContentHandler. Los argumentos son, por orden, el valor entre las etiquetas inicio y fin del elemento, inicio y extensión del elemento.

warning

-warning(SAXParseException)

Método de la clase ErrorHandler que se lanza al percibir un aviso en el proceso de parseo. Recibe como argumento la excepción que lo lanzó.

error

-error(SAXParseException)

Método de la clase ErrorHandler que se lanza al percibir un error en el proceso de parseo. Recibe como argumento la excepción que lo lanzó.

fatalError

-fatalError(SAXParseException)

Método de la clase ErrorHandler que se lanza al percibir un error fatal en el proceso de parseo. Recibe como argumento la excepción que lo lanzó.

getValue

-getValue(int)

Método de la clase Attributes que nos devuelve el valor del atributo cuya posición es el parámetro recibido y lo devuelve en String.

10.2 Comandos

10.2.1 PSQL

Arrancar psql

-psql [options] [dbname [username]]

Arranca psql y se conecta a la base de datos con el usuario especificados en los parámetros. Puede arrancarse psql con muchas opciones diferentes.

Conectarse a una base de datos

-\c database_name

Se conecta a la base de datos especificada.

Ver ayuda

-\h

Muestra la ayuda relacionada con SQL

-\?

Muestra la ayuda relacionada con los comandos específicos de psql.

10.2.2 Indexador de Lucene

El comando para ejecutar el indexador de Lucene es el siguiente:

java -jar MedLucSAX.jar *arg1 arg2 arg3*

Donde los argumentos son:

- arg1→ String de la ruta absoluta del directorio que contiene los archivos .XML de MEDLINE.

- arg2→String con la ruta absoluta del directorio en el que se guardará el index.
- arg3→ String cuyo valor es “true” si queremos que la base de datos sea sobrescrita/creada o “false” si lo que queremos es añadir nuevos documentos a un index existente.

Para asignar más memoria a la máquina java utilizamos las opciones de ejecución –Xms y –Xmx (por ejemplo, -Xms512M -Xmx512M).

10.2.3 Indexador de PostgreSQL

El comando para ejecutar el indexador de Postgresql es:

java -jar MedSQLSAX.jar *arg1 arg2 arg3 arg4 arg5*

Donde los argumentos son de tipo String y son:

- arg1→ Ruta absoluta del directorio que contiene los archivos .XML de MEDLINE.
- arg2→Usuario con el que debe conectarse la aplicación.
- arg3→El password de dicho usuario.
- arg4→Nombre de la base de datos que contendrá los datos.
- arg5→ String cuyo valor es “true” si queremos que la base de datos sea creada/borrada-creada o “false” si lo que queremos es añadir nuevos documentos a una base de datos existente.

Para asignar más memoria a la máquina java utilizamos las opciones de ejecución –Xms y –Xmx (por ejemplo, -Xms512M -Xmx512M).

10.2.4 Motor de búsqueda de Lucene con entorno gráfico

El comando para ejecutar este motor de búsqueda es el siguiente:

java -jar MotorBusquedaLucene.jar

No recibe argumentos. Si se quiere asignar más memoria a la máquina java para la ejecución de este comando basta utilizar las opciones –Xms y –Xmx (por ejemplo, -Xms512M -Xmx512M).

10.2.5 Motor de búsqueda de Lucene textual

El comando de ejecución es el siguiente:

java -jar MotorBusqLucTexto.jar *arg1 arg2 [arg3] [arg4] [arg5] [arg6] [arg7] [arg8] [arg9]*

Los argumentos 1 y 2 son obligatorios pero el resto son opcionales y servirán para indicar que campos queremos que se muestren del resultado de la query. Todos los argumentos son Strings y son:

- arg1→ Ruta absoluta del index de Lucene sobre el que queremos que se realice la búsqueda.
- arg2→ Expresión que representa la query en sintaxis de QueryParser de Lucene.
- [arg3] [arg4] [arg5] [arg6] [arg7] [arg8] [arg9]→ Estos argumentos son opcionales y sirven para indicar que campos queremos visualizar de los resultados. Cada argumento puede tomar 7 valores distintos cada uno correspondiente a un campo posible. Estos valores son "pmid", "article title", "abstract", "pub year", "journal title", "mesh mayor" y "mesh minor".

Para asignar más memoria a la máquina java utilizamos las opciones de ejecución –Xms y –Xmx (por ejemplo, -Xms512M -Xmx512M).

10.2.6 Motor de búsqueda de PostgreSQL con entorno gráfico

Este es el comando:

java -jar MotorBusqSQL.jar

No recibe argumentos. Si se quiere asignar más memoria a la máquina java para la ejecución de este comando basta utilizar las opciones –Xms y –Xmx (por ejemplo, -Xms512M -Xmx512M).

10.2.7 Motor de búsqueda de PostgreSQL textual

Este es el comando de ejecución:

java -jar MotorBusqSQLTexto.jar *arg1 arg2 arg3 arg4*

Donde los argumentos son de tipo String y son:

- arg1→ El usuario con el que debemos conectarnos a la base de datos.
- arg2→La contraseña del usuario especificado en el primer argumento.

- arg3→Nombre de la base de datos sobre la que se debe realizar la búsqueda.
- arg4→ Query a realizar en sintaxis SQL.

Si se quiere asignar más memoria a la máquina java para la ejecución de este comando basta utilizar las opciones –Xms y –Xmx (por ejemplo, -Xms512M -Xmx512M).

Nosotros, Fabián Fernández García y Moisés Azancot Chocrón, autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, y no comerciales, la presente memoria, así como el código, la documentación y/o los prototipos desarrollados relativos a este proyecto.

Firmado:

Fabián Fernández García Moisés Azancot Chocrón

Á Madrid, 20 de septiembre de 2007.